Adat és kiértékelési függőségi elemzés funkcionális nyelvekre – Erlang programok statikus elemzése

Tóth Melinda

Eötvös Loránd Tudományegyetem Informatika Doktori Iskola
Csuhaj Varjú Erzsébet, DSc., habil
doktori iskola vezető

Az informatika alapjai és módszertana oktatási program
Horváth Zoltán, PhD, habil
programvezető

Programozási Nyelvek és Fordítóprogramok Tanszék
Horváth Zoltán, PhD, habil
témavezető

2018

Tartalomjegyzék

1. fejezet

Bevezetés

A doktori kutatásom és ezzel együtt ezen dolgozat célja az volt, hogy az Erlang nevű funkcionális programozási nyelvhez adatfolyam és adatfüggőségi relációkat adjak meg és ezzel lehetőséget teremtsek az erre épülő statikus elemzések definiálására és azok megvalósítására.

Statikus elemzésnek tekintjük azt a módszertant, mely a forráskódok futtatása nélkül[1] elemzi a szoftverek forrásállományait. Célja szerteágazó lehet. Többek között lehet a hibakeresés, kódmegértés támogatása, a kód bonyolultságának, stílusának a mérése vagy akár refaktorálás is. Egy statikus elemző eszközkészlet egy-egy programozási nyelvhez igen széleskörű funkcionalitást kell lefedjen. Az elemzés alkalmazásától függően szükség lehet függvényhívások elemzésére, végrehajtási utakra, szintaktikus-szemantikus összefüggések felderítésére. Az én dolgozatom ezek közül az adatok statikusan követhető elemzésére fókuszál és ennek egy alkalmazására a párhuzamosítható minták felismerésén keresztül. Ez utóbbi esetben sem elegendő csupán az adatok közti összefüggés felderítése. Szükségünk van például a vezérlésfolyam gráfból számítható végrehajtási utak fogalmára is [Boz18].

A dolgozatom a tudományos közélet által elfogadott közleményeken alapuló összefogó mű, mely röviden bemutatja az elért eredményeket és a függelékben összesíti az eredményeket bemutató publikációkat.

A dolgozat bevezető fejezetében röviden ismertetem a kutatási téma indo-

[1] un. fordítási időben

koltságát és a statikus elemzés jelentőségét a szoftverfejlesztésben (1.1. fejezet). Ezután egy rövid ismertetőt adok az Erlang programozási nyelvről (1.2. fejezet) és a RefactorErl keretrendszerről (1.3. fejezet), melyben a dolgozatban bemutatott elemzések meg lettek valósítva. Ezután összefoglalom a dolgozat fő téziseit (1.4. fejezet). Végül röviden ismertetek releváns kapcsolódó irodalmi eredményeket (1.5. fejezet).

Ezt követően egy-egy külön fejezetben foglalom össze a három tézishez tartozó eredményeket. Először Erlang programok adatfolyam elemzéséről írok a nulladrendű, kontextus független elemzéstől indítva, az elsőrendű elemzésen keresztül, egészen a konkurens adatfolyam megadásáig (2. fejezet).

Ezt követően a 3. fejezetben definiálom Erlang programok viselkedés függőségi gráfját és egy relációt, mely az adatok közötti függőség kiszámítására használható.

A harmadik tézishez tartozó 4. fejezet azt foglalja össze, hogyan lehet a statikus elemzést (magába foglalva a korábban bemutatott adatfolyam és adatfüggés relációkat is) párhuzamosítható kódrészletek felismerésére használni.

A dolgozat egy magyar és egy angol nyelvű összefoglalóval zárul (5. és 6. fejezetek), melyet a hivatkozott művek és a saját publikációk kettébontott listája követ. Az első lista tartalmazza a tézisekhez szorosan kapcsolódó publikációkat. A második lista pedig a megjelent további tudományos művek listáját.

A dolgozat függelékében a tézisek leírása során hivatkozott publikációk szerepelnek, melyek a tézisekhez tartozó elemzések részletes leírását tartalmazzák (A., B. és C. függelékek).

1.1. A kutatás jelentősége

A szoftverfejlesztést támogató eszközök jelentősége megnőtt az utóbbi évtizedekben. A szoftverek mérete ugyanis akkorára nőhet, melyet már nehéz átlátni, megérteni, karbantartani. Így a szoftveres támogatás szükségessé vált. A leggyakrabban használt programozási nyelvekhez számos eszköz elérhető [SCI18, Gra18, KOHB15, PBKC18, Ora18, Cpp18]. A szoftverfejlesztést támogató eszközök készülhetnek ugyan általános céllal nyelvfüggetlen módon [RBS13], ugyanakkor úgy tudunk nagy pontosságot elérni az elemzésekkel, ha azok nyelv-

specifikusan, az adott nyelv szemantikájának megfelelően készülnek. Kutatásom fő célja az Erlang nyelv szemantikájához illeszkedő statikus elemzések definiálása, megvalósítása és azok felhasználása a programfejlesztést támogató különböző területeken.

Az Erlang egy dinamikusan típusos programozási nyelv, mely nyelvi szinten támogatja a konkurens programok írását és dinamikus konstrukciók használatát. Így sok kifejezés értéke statikusan, fordítási időben nem ismert, így a hozzá tartozó jelentés sem adható meg. Kutatásom célja, hogy a nyelv dinamikus szemantikájának megfelelő statikus elemzéseket definiáljak. Így az elemzések a valós időben előforduló eredmények egy közelítését adják majd, de előnyük, hogy a forráskódok futtatása nélkül teszik ezt.

Kutatásom fókuszában az Erlang programokban előforduló adatfolyamok és függőségek definiálása áll. Az adatok fordítási idejű ismerete már önmagában is hasznos. Ez lehetőséget nyújt olyan kérdések megválaszolására, hogy eljuthat-e a program adott pontjára egy érték, kideríthetjük egy változó lehetséges értékeinek a halmazát, vagy meghatározhatjuk, milyen üzenetek érkezhetnek egy folyamat üzenet fogadó kifejezésének mintáiba. Ezen kívül az adatfolyam elemzés lehetőséget nyújt a dinamikus konstrukciókban használt kifejezések értékeinek a kiderítéséhez, így lehetőséget teremtve például egy dinamikus hívás felderítésére, egy szerver nevének a kiderítésére vagy refaktorálások biztonságos végrehajtására.

Az adatfolyam kiterjesztéseként előálló adatfüggőség reláció jelentősége abban áll, hogy nem csak azt tudjuk nyomon követni, hogy az egyes adatok hol jelennek meg még a programban, hanem azt is, hogyan függnek egymástól a különböző kifejezések. Így a program egy adott pontján el tudjuk dönteni például azt, hogy az adott kifejezés kiértékelhető-e a másik kifejezés értékének ismerete nélkül, azaz függetlenek-e. Ez különösen hasznos lehet párhuzamosítható komponensek keresésénél. A függés ismerete pedig akkor lehet hasznos, ha azt szeretnénk nyomon követni, hogy egy kifejezés módosítása milyen más kifejezésre van hatással vagy ha meg akarjuk érteni egy kifejezés működését, akkor milyen más kifejezéseket kell még megismernünk.

A szoftverfejlesztést támogató eszközök lehetőséget nyújtanak a mindennapos programozói tevékenységek megkönnyítése mellett arra is, hogy a humán

számára túlságosan nagy méretű forráskódokban olyan összefüggéseket keressen, melyet a fejlesztő már nem képes megtalálni vagy éppen túlságosan időigényes lenne számára. Egy ilyen jellegű feladatra példa a programok optimalizálása párhuzamosítással. Nagy kódbázisban már nem könnyű feladat megtalálni azokat a pontokat, amelyeket párhuzamosítani lehet. Kutatásomban így azzal is foglalkoztam, hogyan lehet párhuzamosítható mintákat definiálni statikus elemzésen alapuló relációk segítségével, ezzel lehetőséget teremtve a mintáknak megfelelő kódrészletek automatikus felismerésére.

Jelen dolgozatban bemutatott definíciókhoz és statikus elemzésekhez tartozó algoritmusok meg lettek fogalmazva a RefactorErl statikus elemző keretrendszerre építve is. Az Erlang nyelv szemantikája alapján megfogalmazott elemzéseket így validnak tekintjük azáltal, hogy az előálló gráfok és az algoritmusok RefactorErl-beli megvalósításának eredményei helyesek voltak.

1.2. Az Erlang nyelv

Az Erlang [CT09, LMC10] egy dinamikusan típusos, szigorú kiértékelésű, funkcionális programozási nyelv, melyet a 80-as években kezdtek fejleszteni telekommunikációs rendszerekhez. Innen jön a nyelv legnagyobb előnye is, hogy nyelvi szinten támogatja a konkurenciát és az elosztottságot.

Az Erlang programozási nyelvet gyakran emlegetik mint Erlang/OTP, azaz Erlang Open Telecom Platform. Ennek ugyan csak történelmi jelentősége van, ám azt mégis jól leírja, hogy nem csak egy programozási nyelvről beszélünk, hanem a hozzá tartozó virtuális gépről, keretrendszerről, széles könyvtári támogatásról, tervezési mintákról stb. is. Az Erlang/OTP a hibatűrő, megbízható, masszívan konkurens, valós idejű rendszerek fejlesztésének egy kiváló eszköze. Éppen ezért Erlangban is jelentős számú szoftver íródott már, ami azt jelenti, hogy a sok millió sornyi szoftverek karbantartását és fejlesztését segítő szoftverek ehhez is szükségesek.

Az Erlang programok modulokba szervezett függvény definíciókból állnak. Egy-egy modulban a függvényeken kívül még un. attribútumok is megjelenhetnek, melyek a modul interfészét, előfeldolgozó (preprocesszáló) utasításokat, rekordokat stb. definiálnak.

Az alábbi kódrészlet egy egyszerű modul definícióját mutatja:

```
-module(simple).
-export([simple/1]).

simple([]) -> 0;
simple([H|T]) when H > 0->
    SubResult = simple(T),
    1 + SubResult.
```

A `simple` modul a nevét és interfészét deklaráló attribútumok (`module`, `export`) után egy függvényt definiál, melyet az Erlang virtuális gépből az alábbi módon tudunk meghívni: `simple:simple([1,2,3])`. Erlangban egy függvényt a definiáló modul neve, a függvény neve és az aritása (azaz a paramétereinek a száma) azonosít, ezért a függvényekre hivatkozni az alábbi módon tudunk: `mod:fun/arity`. Amennyiben a modul egyértelmű, akkor azt elhagyhatjuk.

Az Erlang függvények több ágból állhatnak. Minden ágban definiálunk egy minta sorozatot paraméterként (lehet üres is, azaz a paramétereinek a száma nulla), egy opcionális őrfeltételt a `when` kulcsszó után és egy kifejezés sorozatból álló törzset. A `simple/1` függvény két végrehajtási ággal van definiálva és egy listát vár argumentumként. Amennyiben a kapott paraméter egy üres lista, akkor nullát ad eredményül. Amennyiben a listának legalább egy eleme van és az első elem egy pozitív szám, akkor rekurzívan meghívja önmagát a lista farkára, majd az eredményhez hozzáad egyet. A második ágban egy két kifejezésből álló szekvenciát láthatunk. A függvény értéke a törzsében lévő utolsó kifejezés értékével egyezik meg. A `simple/1` függvény definíciója a listában lévő pozitív értékeket számolja meg, amennyiben a bemeneti lista egy számokat tartalmazó lista.

Az alábbiakban összefoglalom a nyelv néhány tulajdonságát, melyek fontosak a statikus szemantikus elemzések szempontjából:

- egy változó értéke nem változhat, azaz egyszeres értékadású nyelvről van szó;
- mintaillesztést több formában is használhatunk:

  - változókhoz értéket köthetünk,
  - összetett adatszerkezetek elemeire bonthatunk,
  - vezérelhetjük a végrehajtást;

- minden kifejezés (még az elágazó szerkezetek is), azaz mindennek van értéke;
- mintákban mintákat is definiálhatunk;
- vannak statikus és dinamikus függvényhívások is. Így nem kell konstans értékeket írnunk a függvényhívásban a függvény nevének azonosítójaként, hanem elegendő az, hogy a kifejezés értéke futási időben atomra[2] értékelődjön ki. Mivel minden kifejezésnek van értéke, ezért valójában tetszőleges kifejezés állhat függvény nevének a helyén.;
- a konkurencia elemei, az üzenet küldés és fogadás is, kifejezései a nyelvnek;
- a nyelv csak néhány alaptípust (number (int és float), atom, pid, port, reference, function) és összetett adatszerkezetet (tuple, list, binary, map) definiál, további új típus definiálására nincs lehetőség.

Az Erlang nyelv szintaxisának egy megadása az A. függelékben található cikk A. függelékében olvasható. A dolgozatban foglaltak ezt veszik majd alapul.

1.3. RefactorErl

Erlang esetében különösen nagy jelentősége lehet egy statikus elemzőnek, mely a szigorú statikus típusrendszer hiánya nélkül könnyen leforduló, ám nem feltétlenül jól működő szoftver komponensekben segít hibákat keresni, információkat, összefüggéseket felderíteni.

Az ELTE Informatikai Karán 2006 óta működik a RefactorErl nevű kutatócsoport, mely Erlang programok statikus elemzésével és refaktorálásával foglalkozik. A fejlesztett RefactorErl [HLK+09a, BHH+11, KCH+08] nevű elemző eszköz

[2]Erlangban az atomok lényegében olyan konstans karaktersorozatok, melyeket jellemzően azonosításra használunk.

célja az induláskor az volt, hogy biztonságos forráskód transzformációkat támogasson. Ehhez azonban egy mély szemantikus elemző infrastruktúrát kellett kiépíteni. Ezt az elemző környezetet vette alapul és fejlesztette tovább a kutatócsoport abba az irányba, hogy az eszköz egy iparban is használható kódmegértést, fejlesztést és karbantartást támogató eszközzé váljon. A RefactorErlt sikerrel használták több millió soros Erlang szoftvereken is [TO15].

A dolgozatban bemutatott elemzések a RefactorErl alapelemző infrastruktúrájának a részeként (adatfolyam elemzés), illetve arra építve lettek megvalósítva (mintafelismerés).

1.3.1. Köztes reprezentáció

A statikus elemzés egy szükséges eszköze a forráskód megfelelő reprezentációja, melyen az elemzéseket el lehet végezni. Ezeket a reprezentációkat a forráskód egy köztes reprezentációjának nevezzük, mivel az a forrásszöveg és a gépi kód között van. Elvárásaink az alábbiak a reprezentációval szemben: (i) kellőképpen sok információt kell megőrizzen a forrásszövegről, mivel a refaktorálások után a forráskódot vissza kell tudnunk adni szöveges formában is, (ii) ugyanakkor könnyen bejárható és kereshető adatstruktúra legyen. Ennek megfelelően a RefactorErl egy un. Szemantikus Programgráfot (továbbiakban SPG az angol Semantic Program Graph elnevezésből) épít fel a forráskód és az elemzett információ tárolására. Az SPG egy irányított, élcímkézett gráf, mely három rétegből áll:

- lexikális réteg – tartalmazza a forráskód tokenjeit a whitespace információkkal együtt. A token csúcsok hozzá vannak kötve a hozzájuk tartozó szintaktikus elemekhez.
- szintaktikus réteg – a forráskód szintaxisfáját reprezentálja. A szintaktikus egységek a hierarchikus felépítésük sorrendjében össze vannak kötve egymással.
- szemantikus réteg – a szemantikus elemzések eredményét tárolja. Az egyes szemantikus csúcsok egymással és a hozzájuk tartozó szintaktikus entitásokkal is össze lehetnek kötve.

1.3.2. Elemzések

A RefactorErl egy saját, könnyen kiterjeszthető, módosítható nyelvtan leírásból generálja a lexikális és a szintaktikus elemzőjét. Ezek az elemzők építik fel az SPG első két rétegét. Ezután egy asszinkron elemző infrastruktúra formonként[3] párhuzamosan végrehajtja az alap szemantikus elemzéseket az adott form szintaxisfájára egy előre megadott sorrendben. Az alap elemzések mind inkrementálisak, azaz a szintaxisfa változásának a függvényében (beszúrás, törlés, frissítés) állítják helyre a szemantikus réteg konzisztenciáját a teljes forráskód újraelemzése nélkül. A magasabb rendű elemzések az alap elemzések után futnak majd le, igény szerint, jellemzően a felhasználó kérésére.

Alap elemzések

Az SPG az alábbi szemantikus elemzések által előállított információkat tartalmazza az alkalmazásuk sorrendjében:

- kifejezés kontextus elemző
- változó elemző
- modul elemző
- függvény elemző
- rekord elemző
- adatfolyam elemző (a közvetlen élek számítása)
- specifikáció és típus elemző.

Ezen kívül vannak még technikailag ide tartozó elemzések, melyek az SPG konzisztenciájának a fenntartásáért felelősek, de nem szemantikus elemzések. Ilyenekre példák a forrásszöveg elrendezéséért [4] és a formok szövegének a tárolásáért (optimalizációs szempontból) felelős komponensek.

[3] Az Erlang modulok ponttal elválasztott elemei: függvények és attribútumok.
[4] pretty printer

Magasabb-rendű elemzések

Magasabb-rendű vagy utóelemzések közé azokat az elemzéseket soroljuk, melyek végrehajtásához szükség van a teljes forráskódra felépített SPG-re. Ilyen például egy változó lehetséges értékeinek a kiszámítása, a dinamikus függvényhívás elemzés, konkurens adatfolyam elemzés stb. Ezen elemzések egy másik közös jellemzője, hogy nem inkrementálisak, azaz a forráskód változásával újra kell őket számolni.

1.3.3. Az eszköz funkcionalitásai

A RefactorErl lehetőséget nyújt a forráskódon jelentés megőrző transzformációk végrehajtására [KCH+08]. Több mint 20 általános transzformációt nyújt az alap eszköz, de ezen felül különböző kiterjesztések révén (mint a PaRTE[5] [BFH+14]) még több célzott transzformáció is használható. A refaktorálások során a forráskód változatlan részeinek külalakja megőrződik és az SPG konzisztenciája automatikusan helyreáll.

A refaktorálások mellett számos olyan funkcionalitást is nyújt az eszköz, mely segíti a fejlesztőket a kód megértésében, hibakeresésben. Ennek egyik eszköze egy lekérdező nyelv [HTL10, TBH10a, HBKT11], mely lehetőséget nyújt a felhasználónak a nyelv fogalomrendszerén keresztül kérdések, szemantikus összefüggések megfogalmazására. Ezen a lekérdezőnyelven keresztül elérhetőek olyan komplex elemzés eredményei, mint a dinamikus függvényhívások, mellékhatások számítása vagy az adatfolyam, de lehetőséget ad a forráskód bonyolultságának a lekérdezésére is metrikákon keresztül.

A forráskódban rejlő kapcsolatok feltárásának egyik eszköze a függőségi elemzés, mely különböző szinteken segít a kód összefüggéseinek a megértésében. A modul csoport, modul vagy függvény szinten lévő függőségeket a szöveges kiírás mellett vizuálisan is megjelenti, statikus és dinamikus nézetekben egyaránt [KBT18].

A függőségek vizualizálása mellett a konkurens programok viselkedésének elemzése érdekében folyamatok kommunikációs modelljét is felépíti és megjelení-

[5]ParaPhrase Refactoring Tool for Erlang – automatikus párhuzamosítást támogató komponens

ti [TB14, BT16, IM14]. Illetve a jól definiált viselkedések (generikus szerverek, felügyelő folyamatok, állapotgépek) modelljét is képes kinyerni a forráskódból és vizualizálni a kódmegértés segítése végett [BT16, IM14, BKT18, LTB18, LT18, BST18].

Az eszköz duplikáltkód elemző funkcionalitása segít a forráskódban a kód másolatok felderítésében, eliminálásában és az esetleges duplikátumokból eredő hibák javításában [FT16, FT14, FTK14, FT15b].

Az eszköz program szeletelésen alapuló komponense lehetőséget nyújt a forráskód változásainak nyomon követésére oly módon, hogy a változás hatását számolva kiszűri a változtatás által érintett teszteseteket, azaz kiválasztja, melyeket kell lefuttatni a biztonságos működés ellenőrzéséhez [HBT14, FBT17].

Az eszköz párhuzamosítható mintákat felismerő komponense PaRTE néven ismert [BFH+14, BFH+15, TBK17, KTBH16, KTB18], ám lényegében csak a RefactorErl elemző infrastruktúrájára épülő újabb funkcionalitásról van szó.

Az eszköz számos módon használható. Beépíthető a fejlesztési folyamatba a fejlesztő környezetbe integrálva, de különálló asztali alkalmazásként, vagy éppen böngészőben egy webszerveren keresztül is használható.

1.4. Tézisek

A doktori dolgozat három tézis köré épül:

- **1. tézis** Definiáltam Erlang programok elsőrendű kontextusfüggő adatfolyam-gráfját és adatfolyam relációját az Erlang nyelv szemantikájának megfelelően. A reláció felhasználásával megadtam a konkurens Erlang programok aszinkron üzenetváltásai által meghatározott adatfolyamot. A RefactorErl keretrendszerben megadtam a gráf és a reláció számításának algoritmusát.

- **2. tézis** Definiáltam Erlang programok viselkedés függőségi gráfját és a gráfon értelmezett adatfüggőségi relációt. A RefactrorErl keretrendszerben megadtam a függőség számítás algoritmusát.

- **3. tézis** Megadtam párhuzamos programozási mintáknak megfelelő szekvenciális kódrészletek viselkedését statikus elemzéssel kifejezhető relációk függvényében. A definícióknak megfelelő algoritmusok a RefactorErl keretrendszerben felhasználhatóak párhuzamosítható minták felismerésére.

Ezen dolgozatban megfogalmazott statikus programelemzési definíciók algoritmusait megfogalmaztuk a RefactorErl keretrendszer Szemantikus Programgráfjára építve. Ezen algoritmusok megvalósításait egyrészt teszteltük [TTB+12], másrészt pedig további alkalmazásokba beépítettük, ahol a felhasználás során az elvárt eredményt adták. Így a megfogalmazott elemzéseket validáltuk.

1.5. Kapcsolódó irodalom

A meglévő és fejlesztés alatt álló szoftverek számának rohamos növekedésével egyre nagyobb jelentőséggel bírnak a forráskód elemzők. Számos módszer és ezekhez adott eszköz létezik, melyek különböző szempontok szerint csoportosítását és vizsgálatát mutatják be az alábbi cikkek [Bin07, GS15, EN08, Wög05, KA10]. A statikus elemzés formális módszertanának egy alapozó művének a Nielson et al. könyv [NNH05] tekinthető.

Érdemes azonban megemlíteni, hogy a statikus programelemzés lényegében tekinthető a fordítóprogramok elemzőkészletének egy variánsának. A [Muc97] könyvben a fordítóprogramok statikus elemző algoritmusai vannak bemutatva.

Mindezek mellett a dolgozatban bemutatott adatfolyam elemző módszert nehéz összevetni a szakirodalomban fellelhető algoritmusokkal. Az általános adatfolyam elemző algoritmus ugyanis megszorítások és/vagy egyenlőségek megoldásával számolja egy-egy blokkból kifolyó vagy befolyó adatok lehetséges értékeit. Valamint ezt a vezérlésfolyam gráf bejárásával és építésével együtt teszi meg. Az Erlang programokhoz definiált statikus elemzéseink kettéválasztják az adat- és vezérlésfolyam elemzést. Minkét elemzés a nyelv szintaktikus kategóriáit figyelembe véve definiálja az közvetlen adat- és vezérlésfolyamot tartalmazó adatfolyam-gráfot és vezérlésfolyam gráfot. Az közvetlen elérések kiszámítása pedig a gráfon értelmezett relációk segítségével adhatóak meg, melyek speciális gráfbejárásokkal algoritmizálhatóak.

A dolgozatban bemutatott elemzések a RefactorErl rendszerben lettek megvalósítva. A bemutatott RefactorErl statikus elemző eszköz mellett Erlanghoz is több elemző eszköz érhető el. Ezeket az eszközöket nem egymás versenytársaiként kell kezelni, inkább kiegészítik egymást.

A Dialyzer [LS04, CS11a, CS11b] egy statikus elemzésen alapú hibakereső eszköz. Elemzéseinek fókuszában a statikus típusozással[6] [LS06] kiszűrhető hibák felderítése áll. Az elemzéseket az Erlang magnyelvén, Core Erlangon [Car01] definiálja. Konstans értékek propagálásához, és változók értékeinek a kiderítéséhez a Dialyzer is implementál egy „ad-hoc" adatfolyam elemzést belső használatra. A Dialyzer a tipikus hibák azonosításával segíti a fejlesztőket. A RefactorErl eszközei azonban arra fókuszálnak, hogy lehetőséget biztosítsanak a fejlesztőknek arra, hogy saját magunk meg tudják fogalmazni a kérdéseiket és ezzel felderítsék a váratlan hibák helyét is.

Erlanghoz kapcsolódóan meg kell említenünk a Wrangler nevű statikus elemző és refaktoráló eszközt is [LT08]. Az eszköz széleskörű rafaktoráló eszközkészletet biztosít Erlanghoz, és támogatja például a duplikált kódok megtalálását és eliminálását [LT11]. Folyamatok refaktorálásához a Wrangler is használ adatfolyam és adatfüggőség fogalmakat, bár ezek algoritmusa nem publikált [LT15, LTOT08].

Adatfolyam-gráf építése nélkül határoz meg adat függőségi éleket a Slicerl [STT12] eszköz. Az eszköz nem számol/tárol olyan részletességgel információt mint a RefactorErl, a konstrukciókat magasabb szinten kezeli. Emellett nem különbözteti meg a valós függőségi és másolás éleket, ezért közvetett adatfolyam reláció nem számolható a reprezentációján.

Hasonlóan az adatfolyam elemzés területéhez, a párhuzamosítás támogatása statikus és dinamikus elemzéssel sem csak Erlang programok esetén merül csak fel problémaként. Többféle megközelítés létezik. A párhuzamosítást végezheti a fordító maga; vagy egy külön refaktoráló eszköz automatikusan; vagy éppen csak a potenciális jelöltekre mutathat rá egy eszköz és a fejlesztő választja ki, hogy mit és hogyan transzformál. Mindegyik módszer hasznos lehet.

Funkcionális nyelvekhez több automatikusan párhuzamosító fordító is adott. Ilyen például a Data Parallel Haskell [CLPJ+07] és a SkelML [MIK97]. Az utóbbi a tipikus magasabb rendű függvények mentén keresi a jól párhuzamosíthatóakat.

[6]Erlang esetén ezt "success typing"-nak nevezik.

Az általam adott mintafelismerés tetszőleges rekurzív függvénnyel leírt jelölteket is meg tud találni. Ezen kívül célunk nem csak automatikus párhuzamosítás volt a PaRTE eszközzel [BFH+14], hanem a jelöltek megtalálása után a fejlesztőnek ajánlatokat tenni a párhuzamosításra, és hagyni a fejlesztőt mint a szoftver leginkább ismerőjét meghozni a döntést a párhuzamosításról.

Számos eszköz létezik nem funkcionális nyelvekhez is, mely a párhuzamosítást segíti elő [SV12]. A mi módszerünk azonban abban nyújt mást, hogy statikusan elemzett forráskódok alapján nyújt jelölteket a strukturált párhuzamosság bevezetésére. Ehhez hasonlóan, Hammacher et al. [HSHZ09] által javasolt megközelítés is jelölteket ad a fejlesztőknek, és rangsorolja őket a legnagyobb nyereség tekintetében. Azonban ez az elemzés dinamikus függőségeket használ fel.

2. fejezet

Adatfolyam elemzés

Az adatfolyam elemzés célja, hogy információt adjon arról, hogyan manipulálja egy program az adatait, illetve melyek a lehetséges értékei a különböző kifejezéseknek/változóknak a program egy adott pontján. Kutatásom fő célja az volt, hogy definiáljam Erlang programokhoz az adatelérés relációt és az ebből származó információt különböző statikus elemzésekben felhasználhatóvá tegyem. Az adatfolyam elemzés az un. adatfolyam-gráfra épül (továbbiakban DFG az angol Data-Flow Graph kifejezés rövidítéséből), mely az Erlang programok kifejezései között előforduló közvetlen adatfolyamot reprezentálja. A programban előforduló összes (közvetlen és közvetett is) adatfolyamot, pedig a gráfra épülő adatfolyam reláció definiálja majd. Attól függően, hogy az elemzés mennyi futási idejű kontextust vesz figyelembe beszélhetünk nullad-, első- vagy magasabb-rendű elemzésről. A magasabb-rendű elemzés egy példájaként bemutatom a konkurens Erlang programok adatfolyamának elemzését is.

A fejezetben bemutatott nulladrendű elemzés alapja a [LÖ9] cikkben megfogalmazott adatfolyam elemzés. Az ott megfogalmazott adatelérési reláció azonban nem teljesíti a tranzitivitás tulajdonságot, ami az általam végzett kutatásban szükséges volt, ezért ezt módosítottam és kiterjesztettem az ott leírtakat.

2.1. Nulladrendű adatfolyam elemzés

Erlang programok adatfolyam-gráfját egy címkézett, irányított gráfként reprezentáljuk, melynek csúcsai az Erlang program kifejezései és részkifejezései (E), élei pedig a kifejezések közötti adatáramlást írják le (N). Akkor vezet egy él u csúcsból v csúcsba, ha a program végrehajtása során az u csúcshoz tartozó kifejezés értéke belefolyik a v csúccsal reprezentált kifejezés értékébe. Azaz a két kifejezés értéke az adott végrehajtási úton megegyezik. A DFG élei csak a közvetlen adatfolyamot reprezentálják.

$$\text{DFG} = (N,\, E)$$

A adatfolyam-gráfban négy fajta élcímkét vezetünk be:

- $\xrightarrow{\mathbf{f}}$ **(flow él)**: $n_1 \xrightarrow{\mathbf{f}} n_2$ jelentése az, hogy n_2 értéke az n_1 értékének egy másolata lehet.

  Ez az eset áll fenn például akkor, amikor egy függvényhívás aktuális paramétereinek az értékei a formális paraméterekbe másolódnak, vagy amikor egy mintaillesztő kifejezés jobb oldalán álló kifejezésének az értéke belemásolódik a bal oldalon álló mintába. Például az $\{X, Y\} = \{1, 2 + 3\}$ kifejezésben az $\{1, 2 + 3\}$ pár értéke belefolyik az $\{X, Y\}$ mintába.

- $\xrightarrow{\mathbf{c_i}}$ **(konstruktor él)**: $n_1 \xrightarrow{\mathbf{c_i}} n_2$ jelentése az, hogy n_2 egy összetett adat, melynek az i-edik pozícióján lévő részkifejezése n_1.

  Ez az eset jellemzi például a listák, rendezett n-esek létrehozó kifejezéseit. A korábban mutatott kifejezésben például az 1-hez tartozó csúcs $\xrightarrow{\mathbf{c_1}}$ éllel lesz bekötve az $\{1, 2\}$ csúcsába, hiszen a rendezett n-es (pár) első eleme az 1.

- $\xrightarrow{\mathbf{s_i}}$ **(szelektor él)**: $n_1 \xrightarrow{\mathbf{s_i}} n_2$ jelentése a konstruktor él fordítottja mintákra, azaz n_2 az i-edik eleme n_1-nek egy mintában.

  Erre példa a korábbi $\{X, Y\}$ minta, melyből $\xrightarrow{\mathbf{s_1}}$ él mutat majd az X csúcsába, hiszen ennek a mintának az első eleme az X.

- $\xrightarrow{\mathbf{d}}$ **(függőségi él)**: $n_1 \xrightarrow{\mathbf{d}} n_2$ jelentése az, hogy n_2 értéke függ az n_1 értékétől, de nem egyezik meg vele.

  Erre az esetre példa a bemutatott példa $2 + 3$ részkifejezése, ahol is a $+$ csúcsot reprezentáló kifejezés függ a 2 és a 3 csúcsoktól. Értékük nem egyezik meg ugyan, de a kiszámítás során függ tőlük.

Az adatfolyam-gráfot egy kompozícionális szabályhalmaz alapján fel lehet építeni, mivel az csak a közvetlen adatfolyam éleket tartalmazza. Az Erlang nyelv szemantikáját figyelembe véve a szintaktikai elemekhez definiálva van egy szabály, mely azt írja le, hogy amennyiben a szabály bal oldalán található kifejezéssel találkozunk, akkor milyen éleket kell felvennünk az adatfolyam-gráfba a részkifejezéseket reprezentáló csúcsok közé. Ez lehetővé teszi, hogy az adatfolyam-gráfot az SPG bejárásával fel tudjuk építeni. Az adatfolyam-gráfot felépítő szabályok részletes bemutatása az A függelékben található az első cikk 3.1. fejezetében és B. függelékében.

Tekintsük a statikus függvényhívás adatfolyam szabályát példaként (2.1. ábra). A (*Fun. call*) szabály azt írja le, ha találunk egy e_0 kifejezést, mely az m modul g függvényét hívja meg (lokális vagy minősített módon), akkor az i-edik aktuális paraméter értéke belefolyik az $m : g/n$ függvény összes klózának i-edik formális paraméterébe (mintájába): $e_i \xrightarrow{\mathbf{f}} p_i$ tetszőleges $i \in [1..n]$ esetén. Illetve, a függvény visszatérési értéke, azaz a függvény minden klózának utolsó kifejezésének az értéke, belefolyik a kiinduló függvényhívás kifejezésbe: $e_i^j \xrightarrow{\mathbf{f}} e_0$ tetszőleges $i \in [1..n]$ és $j \in [1..m]$ esetén. A statikus adatfolyam szabályok a konkrét végrehajtási út ismerete nélkül épülnek fel, így az összes lehetséges értéket figyelembe veszik.

Mivel a szabályokat a függvényhívás kifejezésre is definiáltuk, így az eredményül kapott adatfolyam-gráf függvények között is kapcsolatot teremt, azaz interprocedurális.

A teljes (közvetlen és közvetett) adatfolyam elérés kiszámítását az úgynevezett adatfolyam reláció definiálásával adjuk meg az adatfolyam-gráfon.

	Kifejezés	Közvetlen DFG élek
(Fun. call)	e_0: $\mathsf{m} : \mathsf{g}(e_1, \ldots, e_n)$ or $\mathsf{g}(e_1, \ldots, e_n)$ m:g/n: $\mathsf{g}(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{g}(p_1^m, \ldots, p_n^m)$ when $g_m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_1 \xrightarrow{\mathbf{f}} p_1^1, \ldots, e_1 \xrightarrow{\mathbf{f}} p_1^m$ $\vdots$ $e_n \xrightarrow{\mathbf{f}} p_n^1, \ldots, e_n \xrightarrow{\mathbf{f}} p_n^m$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_m}^m \xrightarrow{\mathbf{f}} e_0$

2.1. ábra. Függvény hívás DFG szabálya

2.1.1. Nulladrendű Adatfolyam Reláció

Az adatfolyam reláció ($\overset{\mathbf{0f}}{\rightsquigarrow}$) azt definiálja, hogy mikor mondhatjuk, hogy egy kifejezés értéke egy másik kifejezés értékével megegyezik oly módon, hogy az egyik a másik másolata. $n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2$ jelentése, hogy a kifejezés, melyet az n_2 csúcs reprezentálja a DFG-ben, az n_1 másolata.

Az adatfolyam reláció ($\overset{\mathbf{0f}}{\rightsquigarrow}$) az a minimális reláció, mely teljesíti az alábbi tulajdonságokat:

$$n \overset{\mathbf{0f}}{\rightsquigarrow} n \qquad \text{(reflexive)}$$

$$\frac{n_1 \xrightarrow{\mathbf{f}} n_2}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2} \qquad \text{(f rule)}$$

$$\frac{n_1 \xrightarrow{\mathbf{c_i}} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3,\ n_3 \xrightarrow{\mathbf{s_i}} n_4}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_4} \qquad \text{(c-s rule)}$$

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_3} \qquad \text{(transitive)}$$

Azaz a reláció fennáll, (i) egy kifejezésre önmagával (reflexív); (ii) ha egy közvetlen $\xrightarrow{\mathbf{f}}$ él vezet két csúcs között; (iii) ha egy értéket becsomagolunk egy összetett adatszerkezet i-edik pozíciójára, az elterjed valahová és ugyanarról a pozícióról kicsomagoljuk; illetve (iv) tranzitivitás mentén terjed.

Megkülönböztetünk előre és visszafelé haladó adatfolyam elemzést aszerint,

hogy arra van szükségünk, hogy egy érték hová folyik el (előre) vagy honnan érkezett (visszafelé) az adott pontra.

Egy adott n kifejezés ($n \in N$) lehetséges értékeinek a halmazát ($Origin(n)$) és azon kifejezések halmazát, melyekbe potenciálisan belefolyhat az n kifejezés értéke ($Reach(n)$) az alábbi formulák adjuk meg:

$$Origin(n) = \{n_i | n_i \in N, n_i \overset{\mathbf{0f}}{\rightsquigarrow} n\}.$$

$$Reach(n) = \{n_i | n_i \in N, n \overset{\mathbf{0f}}{\rightsquigarrow} n_i\}.$$

Az adatfolyamra épülő további elemzések gyakran ezen halmazok egy részhalmazával dolgoznak, ugyanis arra van szükségük, hogy melyek a konkrét értékek az adatfolyam útvonalak végén. Azaz hol vannak azok a pontok, ahová már nem folyik be adat vagy amiből nem folyik ki adat. Például ilyen a dinamikus-függvényhívás elemzés is, melynek adott kifejezések lehetséges értékeire van szüksége: milyen atom lehet egy függvénynevet reprezentáló kifejezés értéke [HK13]. Ehhez definiáltam a kompakt adatfolyam relációt, mely a korábbi szabályok kiegészítésével kapható.

Nulladrendű kompakt előrehaladó adatfolyam reláció : $\overset{\mathbf{0f_{cf}}}{\rightsquigarrow}$ az a minimális reláció mely teljesíti az alábbi tulajdonságot:

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ \nexists n_3,\ n_3 \neq n_2 :\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{0f_{cf}}}{\rightsquigarrow} n_2} \qquad \text{(f-compact)}$$

Nulladrendű kompakt visszafelé haladó adatfolyam reláció : $\overset{\mathbf{0f_{cb}}}{\rightsquigarrow}$ az a minimális reláció mely teljesíti az alábbi tulajdonságot:

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ \nexists n_0,\ n_0 \neq n_1 :\ n_0 \overset{\mathbf{0f}}{\rightsquigarrow} n_1}{n_1 \overset{\mathbf{0f_{cb}}}{\rightsquigarrow} n_2} \qquad \text{(b-compact)}$$

A nulladrendű adatfolyam reláció elemzése és részletes bemutató példa a viselkedéséről az A. függelékben található az első cikk 3.2. és 3.3. fejezetében.

2.2. Elsőrendű adatelérés reláció

A nulladrendű adatfolyam elemzés kiegészítése az elsőrendű adatfolyam elemzés, mely a függvényhívások kontextusát is követi. Erre azért van szükség, mert a nulladrendű elemzés során belefolyhat olyan érték is egy függvényhívásba, mely nem az adott hívási kontextusból jött. Erre a legjobb példa az identitás függvény és annak két alkalmazása az alábbi kódrészletben.

```
id(X) ->
  X.

calls() ->
  First = id(1),
  Second= id(2).
```

Itt azt várjuk el, hogy a `First` változónak az *Origin* halmazába az `1`, míg a `Second` változó *Origin* halmazába a `2` kifejezés kerüljön be. Azonban a nulladrendű szabályok szerint a függvény formális paraméterébe befolyik az `1` és a `2` is, és a belsejében lévő változó használat miatt a függvény visszatérési értéke megegyezik a paraméterrel. A visszatérési érték belefolyik mindkét függvényhívásba, így a tranzitivitás mentén az eredeti változóink *Origin* halmazába mindkét érték bekerül majd.

Az elsőrendű elemzés ezt a függvényhívási kontextust tárolja el részben az adatfolyam-gráfban a DFG speciális címkézésével, illetve a korábban bemutatott adatelérés relációt új szabályokkal bővíti. A címkézés alapja a hívások indexelése oly módon, hogy a korábbi $\xrightarrow{\mathbf{f}}$ éleket a függvény hívások esetén két újabb élcímkével váltjuk fel:

- $\xrightarrow{\mathbf{call(g,i)}}$ jelöli az i-edik elemzett függvényhívás aktuális paraméterei és a hívott függvény formális paraméterei között mutató élt;
- $\xrightarrow{\mathbf{ret(g,i)}}$ jelöli az i-edik elemzett függvényhívásba visszamutató élt a hívott függvény visszatérési pontjaiból.

A 2.2. ábra mutatja a függvényhívások módosított DFG építő szabályát.

	Kifejezés	DFG élek
(Fun. call)	e_0: $\mathsf{m} : \mathsf{g}(e_1, \ldots, e_n)$ m:g/n: $\mathsf{g}(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{g}(p_1^m, \ldots, p_n^m)$ when $g_m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$. e_0 is the i^{th} analysed call to function $m : g/n$	$e_{l_1}^1 \overset{\mathbf{ret(g,i)}}{\rightarrow} e_0, \ldots, e_{l_m}^m \overset{\mathbf{ret(g,i)}}{\rightarrow} e_0$ $e_1 \overset{\mathbf{call(g,i)}}{\rightarrow} p_1^1, \ldots, e_1 \overset{\mathbf{call(g,i)}}{\rightarrow} p_1^m$ $\vdots$ $e_n \overset{\mathbf{call(g,i)}}{\rightarrow} p_n^1, \ldots, e_n \overset{\mathbf{call(g,i)}}{\rightarrow} p_n^m$

2.2. ábra. Módosított DFG szabálya

2.2.1. Elsőrendű adatelérés reláció

Az elsőrendű adatelérés reláció alapvetően két részből tudjuk felépíteni. Egyrészt minden ami a nulladrendű relációval elérhető az módosított DFG-n (jelölje ezt $\overset{\mathbf{0f'}}{\rightsquigarrow}$), az része a elsőrendű relációnak is (jelölje ezt $\overset{\mathbf{1f}}{\rightsquigarrow}$). Másrészt pedig ki kell egészítenünk a relációt az új $\overset{\mathbf{call}}{\rightarrow}$ és $\overset{\mathbf{ret}}{\rightarrow}$ élek mentén előálló adatfolyamokkal (lásd *call rule* és *return rule*), a tranzitivitást kontextus érzékennyé kell tenni (lásd *call concat. rule*, *return concat. rule*, *reduce rule*) és a konstruktorok mentén lévő adatfolyamot is meg kell adnunk (lásd 1^{st} $c - s$ *rule*).

Az elsőrendű adatelérés reláció részletes elemzése az A. függelékben található (a 3.4., 3.5. és 3.6. fejezetekben), de a definíciót itt is megadom. Az elsőrendű adatfolyam reláció az a minimális reláció, mely teljesíti az alábbiakat:

$$\frac{n_1 \overset{\mathbf{0f'}}{\rightsquigarrow} n_2}{n_1 \overset{\mathbf{1f}[]}{\rightsquigarrow} n_2} \qquad (0^{th} \text{ flow rule})$$

$$\frac{n_1 \overset{\mathbf{c_i}}{\rightarrow} n_2,\ n_2 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_3,\ n_3 \overset{\mathbf{s_i}}{\rightarrow} n_4}{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_4} \qquad (1^{st} \text{ c-s rule})$$

$$\frac{n_1 \overset{\mathbf{call(g,i)}}{\rightarrow} n_2}{n_1 \overset{\mathbf{1f}[\mathbf{call}_{(\mathbf{g,i})}]}{\rightsquigarrow} n_2} \qquad (\text{call rule})$$

$$\frac{n_1 \overset{\mathbf{ret(h,j)}}{\rightarrow} n_2}{n_1 \overset{\mathbf{1f[ret_{(h,j)}]}}{\rightsquigarrow} n_2} \quad \text{(return rule)}$$

$$\frac{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{1f}[\rho]}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{1f}[\mu++\rho]}{\rightsquigarrow} n_3} \quad if\ (\exists f\ \exists i:\ (hd(\rho) = call_{(g,i)}))\ or\ \rho = []$$

(call concat. rule)

$$\frac{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{1f[ret_{(h,j)}}|\rho]}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{1f}[\mu++[\mathbf{ret_{(h,j)}}|\rho]]}{\rightsquigarrow} n_3} \quad if\ (\exists f\ \exists i:\ (last(\mu) = ret_{(g,i)}))\ or\ \mu = []$$

(return concat. rule)

$$\frac{n_1 \overset{\mathbf{1f}[\mu++[\mathbf{call_{(h,i)}}]]}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{1f[ret_{(h,i)}]}}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_3} \quad \text{(reduce rule)}$$

A fenti definícióban az alábbi jelölést használtuk:

- μ jelüli a kontextus listáját;
- $hd(\mu)$ jelöli a kontextus lista fej elemét, $last(\mu)$ pedig az utolsó elemét;
- $\mu + +\rho$ jelöli a kontextus listák összefűzését;
- μ_n pedig a kontextus lista n-edik elemét adja.

Az elsőrendű elemzésről az A. függelék második cikkében is található további példa és tulajdonságok.

2.3. Konkurens adatfolyam elemzés

Az Erlang nyelvi elemekkel támogatja a párhuzamos, konkurens, elosztott programok írását. Kommunikációs modellje az aktor modellnek felel meg. A folyamatok asszinkron üzenetküldésekkel kommunikálnak. Mind az üzenetek küldésére, mind az üzenetek fogadására megvannak a megfelelő szintaktikus nyelvi elemek (a `!` operátor és a `receive` kifejezés), így az adatfolyam gráf építő szabályokat kiegészítettem egy olyan szabállyal, mely az ezek közötti $\overset{\mathbf{f}}{\rightarrow}$ éleket adja (2.3. ábra).

Ez a szabály azt mondja ki, hogy egy üzenetküldő kifejezés jobb oldalán lévő részkifejezését kössük össze egy `receive` kifejezés összes mintájával. A forráskód

	Kifejezés	DFG élek
(Send exp.)	e_0: e_1 ! e_2 e': receive p_1 when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \rightarrow$ $e_1^n, \ldots, e_{l_n}^n$ after $e \rightarrow e_1, \ldots, e_s$ end	$e_2 \xrightarrow{\mathbf{f}} e_0$ $\mathbf{e_2} \xrightarrow{\mathbf{f}} \mathbf{p_1}, \ldots, \mathbf{e_2} \xrightarrow{\mathbf{f}} \mathbf{p_n}$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e', \ldots, e_{l_n}^n \xrightarrow{\mathbf{f}} e'$ $e_s \xrightarrow{\mathbf{f}} e'$

2.3. ábra. Üzenetküldés adatfolyama

konzervatív (lehetséges végrehajtási utakat, valós kontextust nem figyelembe vevő) elemzése során megtehetnénk, hogy minden egyes üzenetküldés jobb oldalát $\xrightarrow{\mathbf{f}}$ éllel hozzákötjük minden egyes `receive` kifejezéshez. Azonban ezzel feleslegesen megnövelnénk a DFG méretét és az adatfolyam relációnk igencsak túlbecsülné a valós adatfolyamot, azaz az *Origin* és *Reach* halmazokat.

Egy pontosítási lehetőség az, hogy a forráskód statikus elemzésével megpróbáljuk eldönteni, hogy milyen folyamatot azonosít az üzenetküldés bal oldalán álló kifejezés. Ennek eldöntéséhez azonban szükségünk van a kifejezés lehetséges értékeinek a kiszámítására, melyhez éppen az adott kifejezésből indított visszafelé haladó adatfolyam reláció szükséges, azaz az $Origin(e_1)$ halmaz elemeinek a kiszámítása. A folyamat azonosítása az alábbi módon történik:

1. meghatározzuk e_1 lehetséges értékeit;

2. ha atomot találunk, akkor megkeressük azokat a kifejezéseket, ahol egy olyan kifejezést használnak egy folyamat regisztrációja során, melynek az értéke megegyezik ezzel az atommal;

3. ha egy Pid-et (process id[1]) találunk, akkor megkeressük, hogy hol lesz ez

[1] a folyamat azonosítója

a Pid egy folyamat létrehozás (`spawn`) eredménye;

4. azonosítjuk a talált (regisztrált vagy nem regisztrált) folyamatot reprezentáló függvényt;

5. megkeressük a függvények törzsében és tranzitív hívási láncában a `receive` kifejezéseket.

Az első négy pontban lévő algoritmusokban mindenhol fel kell használnunk a már korábban definiált $\overset{\mathbf{1f}}{\rightsquigarrow}$ adatfolyam relációt a kifejezések lehetséges értékeinek a meghatározásához. Ez azt jelenti, hogy a DFG üzenetküldésekre vonatkozó élekkel való kiegészítéséhez fel kell használnunk az adatfolyam relációt.

Az adatfolyam reláció felhasználására példaként tekintsük az alábbi formulákat, melyekben az e_1 által meghatározott `spawn` és `register` kifejezéseket határozzuk meg. Mindkét esetben az adatfolyam relációval elérhető *Origin* halmazokat keressük, de leszűrjük a kapott kifejezések típusát arra, hogy azok vajon `spawn` vagy `register` alkalmazás típusúak-e.

- $Spawn(e_1) = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} e_1, type(n) = spawn\_app\}$
- $Reg(e_1) = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} e_1, type(sup(n)) = reg\_app\}$

Az üzenetküldésekből induló adatfolyam élek behúzásának a leírása az A. függelékben található első cikk 3.7. fejezetében található. A folyamatok azonosításával és az Erlang programok kommunikációs modelljének a felépítésével pedig a [Boz18] doktori dolgozat foglalkozik.

2.4. Az adatfolyam elemzés további finomításai

Az adatfolyam elemzés további információk figyelembe vételével tovább pontosítható. Egyrészt beépíthetünk olyan alkalmazásspecifikus információt a DFG-be, mely ugyan a nyelvi elemek szemantikájából nem következik, de az egyes alkalmazások végrehajtási szemantikájából[2] következnek. Egy példa lehet erre a kliens-szerver viselkedés mintát implementáló generikus szerverek használata, ahol az alkalmazások üzleti logikáját az úgynevezett callback modulokban

[2]leginkább az Erlang/OTP viselkedés mintái esetén [LMC10]

implementáljuk. Ezekben az esetekben ugyan nem írunk le üzenetküldő kifejezéseket vagy konkrét függvény hívásokat a `handle_call/3` függvényre, de egy `gen_server:call(To, Msg)` hívás tudottan azt eredményezi, hogy a `To`-val azonosított szerver folyamatban meghívódik a `handle_call(Message, From, State)` fejlécű függvény. Így potenciálisan egy új $\xrightarrow{\mathbf{f}}$ élet adhatnánk az `Msg` és `Message` csúcsai közé a DFG-ben. Ehhez hasonló elemzésekkel bővíthető még a DFG gráf, illetve pontosítható az adatelérés reláció eredménye.

A fentiekben említett és a 2.4.1. és 2.4.2. fejezetekben bemutatásra kerülő elemzéseken kívül pontosításra ad lehetőséget még a vezérlésfolyam gráf alapú végrehajtási utak felhasználása vagy a szimbolikus végrehajtási algoritmusokkal való ötvözés. Az eddig bemutatott elemzés a függvény paramétereibe befolyó értékeket követi a függvény törzsében, de a törzsben lévő döntési pontokat nem építi be az elemzésbe. Adott végrehajtási útra nézve a feltételekben szereplő megszorítások vizsgálata tovább pontosíthatja az eredményt. Ugyanakkor a megfelelő algoritmusok nagy számításigénye a gyakorlati használhatóságot rontja. Az adatfolyam algoritmusokról a 2.5. fejezetben adok összefoglalót.

2.4.1. N-ed-rendű elemzés

Az Erlang funkcionális programozási nyelv jellegéből adódóan természetes nyelvi konstrukciónak tekinthetők a magasabb rendű függvények, melyek további függvényeket kaphatnak paraméterül. Ez az adatfolyam elemzés szemszögéből azt jelenti, hogy egy-egy függvényhívás mentén nem csak az válik szükséges információvá, hogy éppen hányadik hívását elemezzük az adott függvénynek, hanem az is, hogy milyen függvény az, amit az adott függvény megkap argumentumként.

Illusztrációnak tekintsük a 2.4. ábrán bemutatott példát. Amennyiben a `func/2` függvényt úgy hívjuk meg, hogy az `fst/1` függvényt adjuk át neki argumentumnak, akkor a függvényhívás *Origin* halmazába csak az `1`-nek kellene belekerülni, míg az `snd/1` hívás esetén a `2`-nek kell az eredményben lenni.

A kiterjesztett elemzésről további információ az A függelékben található második cikk 5. fejezetében található.

```
func(Fun, Data)->
    Fun(Data).

call_hd()->
    f(fun fst/1, {1,2}).

call_tail()->
    f(fun snd/1, {1,2}).
```

```
fst({A, B})->
    A.

snd({A, B})->
    B.
```

2.4. ábra. Magasabb-rendű függvények használata

2.4.2. Iteratív adatfolyam elemzés

Az adatelérés reláció több más elemzés alapjául szolgál. Erre egy példát már a dolgozat is mutat a konkurens adatfolyam elemzésen keresztül. Ugyanis abban már az adatfolyam relációt használjuk fel a folyamatok azonosítására, mely alapján új éleket kötünk be az adatfolyam-gráfba. Azonban minden egyes él, melyet a DFG-be kötünk, újabb csúcsokat eredményezhet az *Origin* halmazokba, ezzel együtt pedig újabb lehetséges folyamatokat azonosíthatunk. Az új folyamatok új küldés-fogadás párokat azonosíthatnak. Ezt egészen fixpontig iterálhatjuk, hogy minél pontosabb eredményt kapjunk.

Egy másik példa lehet a dinamikus függvényhívás elemzés ([HK13]), mely a konkurens elemzéshez hasonlóan az adatfolyam elemzés segítségével deríti fel a lehetséges függvényazonosító értékeket. Amennyiben ez az elemzés tud új függvényeket azonosítani, akkor a DFG további $\overset{\mathbf{call}}{\rightarrow}$ és $\overset{\mathbf{ret}}{\rightarrow}$ élekkel tud bővülni, ami szintén azt eredményezheti, hogy az adatelérés reláció újabb eredményeket hozhat be a lehetséges értékek *Origin* halmazába.

Azonban ezen a ponton érdemes megemlíteni az elemzések költségének és pontosságának arányát. Egy nulladrendű elemzés lecserélése elsőrendűre akár azt is eredményezheti, hogy gyorsabban tudunk sokkal pontosabb eredményt előállítani (részletekért lásd a 2.5. fejezetet). Az iteratív elemzések költsége azonban mindig növeli a végrehajtási időt, még akkor is, ha a pontosítás alig hoz egy-két új eredményt. Ezért ipari környezetben az iteratív elemzés nem alkalmazott. Az elsőrendű elemzés kiegészítve a konkurens adatfolyammal elégségesnek bizonyult.

2.5. Az algoritmusokról

Az adatfolyam-gráf és a relációk számításának algoritmusát megadtam a RefactorErl keretrendszer eszközkészletére támaszkodva. Az algoritmusok megvalósításakor figyelembe vettem a RefactorErl elemző infrastruktúrájának adottságait. Így a megvalósítás során az adatfolyam-gráf az SPG részeként lett megadva, azaz nem hozok létre új csúcsokat, hanem az SPG meglévő csúcsait fogják az adatfolyam élek összekötni. Az adatfolyam-gráfot a RefactorErl infrastruktúrájának megfelelően formonként/függvényenként párhuzamosan építjük fel. Ekkor kerülnek bekötésre a közvetlen adatfolyam élek. Ez a megvalósítás képes maximálisan kihasználni a környezeti adottságokat. Az algoritmus az SPG részeként előálló szintaxisfa részeket járja be szélességi bejárással és számolja ki a szükséges éleket. Minden függvény szintaxisfáját egyszer.

Szintén a RefactorErl által adott lehetőségeket kihasználva megadtuk az adatfolyam-gráf inkrementális helyreállító szabályait is. Azaz ha egy részfát módosítunk, törlünk, beszúrunk a szintaxisfába, akkor nem kell az egész program (de még a teljes függvényt sem) újraelemezni és a hozzá tartozó adatfolyam-gráf éleket kiszámolni. Elegendő csak a változásnak megfelelően egy helyreállítást végrehajtani.

A nulladrendű és az elsőrendű adatfolyam-gráf építésekor is csak egyszer járjuk be minden függvény szintaxisfáját (közben támaszkodva egyszerű szintaktikus és szemantikus lekérdezésekre), így ezek algoritmusa a szintaxisfa méretével, azaz a forráskód nagyságával arányos. Lényeges különbség a nulladrendű és elsőrendű elemzésben csak az *Origin* és *Reach* halmazok számításában és ezek elemszámában van, azaz az adatfolyam reláció kiszámításában.

Az adatfolyam reláció számításának algoritmusa nem inkrementális, azaz a forráskód változási révén újra kiszámítandó. Míg az adatfolyam-gráf élei, melyek a közvetlen élek, jól számolhatók a lokális változások mentén, addig az adatfolyam reláció elemei potenciálisan a teljes gráf bejárásával, közvetett információk alapján készülnek el, így a lokális változások hatása nem mindig észrevehető/helyreállítható.

Az adatfolyam reláció által meghatározott $Origin(e)$ és $Reach(e)$ halmazokat iteratívan állítjuk elő. Egy lezártat számolunk az e kifejezésből kiindulva az

SPG-n a reláció szabályaival paraméterezve. A nullad- és az elsőrendű elemzés közötti lényeges különbség a μ kontextus lista figyelésében és karbantartásában áll, melynek leginkább hosszú hívási láncok esetén lehet nagyobb költsége.

Az elsőrendű elemzés noha bonyolultabb algoritmussal számítható, mégis gyorsabb tud lenni a nulladrendű elemzésnél. Ennek egyik oka az, hogy a függvényhívások kontextusinformációjának a hiányában az elsőrendű elemzésben a függvény visszatérési pontja elvisz az összes adott függvényre vonatkozó függvényhívás kifejezésbe, majd onnan tovább kell folytatni a bejárást. Ugyanakkor az elsőrendű elemzésben a függvény visszatéréséből csak az aktuális függvényhívásba jutunk vissza, így számos olyan részfát nem kell bejárnunk, amit a nulladrendű elemzés bejár. Ezzel gyorsabb a számítás. Ugyanakkor a pontosság is megnő ugyanezen érvek mentén. A nem bejárt részfákból csak olyan csúcsokat vennénk fel a halmazba, mely a valós végrehajtás során nem lenne elérhető. Ezért az elsőrendű számítás eredményhalmazba jobban közelíti a dinamikus/futási idejű értékeket.

	Előre haladó elemzés (*Reach* halmaz)					
A teszthalmaz elemszáma	1000	10000	5000	10000	5000	5000
Legnagyobb különbség	8.41	9.28	15.87	320.96	2123.53	594.62
Legnagyobb pontosság	19.04%	5.26%	50%	1.78%	0.16%	0.44%
Átlagos különbség	2.30	1.77	1.48	2.85	5.17	3.77

2.1. táblázat. *Reach* halmaz számítása nullad- és elsőrendben

A 2.1. és 2.2. táblázatok a futási idők arányáról mutatnak mérési eredményeket. Az oszlopokban lévő számokat n darab különböző kifejezésre számítva csoportosítva adjuk meg (a kifejezéscsoportok diszjunktak). Az n értéke változott a mérések során, így ennek az értékét mutatja az első sor. A táblázat oszlopaiban ezekre a kifejezéscsoportokra számított *Reach* és *Origin* halmaz előállításának a legjobb és átlagos eredményeit mutatja nagy méretű RefactorErl adatbázisokon. A táblázat soraiban a „különbség” sorok azt mutatják, hogy hányszor volt lassabb a nulladrendű elemzés az elsőrendűnél. A „pontosság”

	Visszafelé haladó elemzés (*Origin* halmaz)					
A teszthalmaz elemszáma	10000	8000	1000	2500	10000	10000
Legnagyobb különbség	45.61	81.81	7.31	51.61	9.12	30.22
Legnagyobb pontosság	4.20%	5.56%	37.20%	7.82%	5.26%	16.67%
Átlagos különbség	2.19	1.85	2.06	2.69	1.73	1.64

2.2. táblázat. *Origin* halmaz számítása nullad- és elsőrendben

sorok pedig azt jelentik, hogy hány százaléka jelenik meg a nulladrendű eredményeknek az elsőrendű eredmények között is. A mérések során elvégeztük annak az invariánsnak az ellenőrzését, hogy az elsőrendű halmazok részhalmazát adják-e a nulladrendű halmazoknak. A tulajdonság mindig fennállt.

Látható, hogy nagyon extrém esetekben több mint kétezerszer lassabban futhat le a nulladrendű elemzés mint az elsőrendű elemzés. Azonban az átlagos értékek 1.48 és 5.17 között mozognak.

Továbbá, a táblázatok pontossági sorai azt is mutatják, hogy az alkalmazástól függően az eredeti nulladrendű eredményhalmaz töredéke lehet a pontosabb elsőrendű eredmény. Szélsőséges esetben akár kevesebb mint 1%-a lehet az elsőrendű elérési halmaz mérete a nulladrendűhöz képest.

2.5.1. Az *Origin* és *Reach* halmazok felhasználás

Gyakorlatban igen nagy jelentőséggel bír az adatfolyam elemzéssel előálló *Origin* és *Reach* halmazok ismerete. A teljesség igénye nélkül kiemelek most pár alkalmazást, további felhasználásokat listáznak a 2.6. fejezetben felsorolt publikációk.

Az *Origin* halmazok egyik legfontosabb jelentősége a kódmegértés támogatásában rejlik. Ezek közül kiemelném azt, hogy a RefactorErl egy lekérdező nyelvet biztosít, mellyel a fejlesztők a forráskódban rejlő szemantikus összefüggéseket tudják kinyerni. A lekérdező nyelvet mutatják be az alábbi publikációk: [TBH10a, HBKT11]. A lekérdezőnyelv egy fontos eleme az *origin* művelet, mellyel egy adott kifejezés lehetséges értékeit listázhatjuk ki. Pél-

dául a @*expr.origin* az adott pozíción lévő kifejezés lehetséges értékeinek a kiszámítására vonatkozó lekérdezés.

Egy összetettebb lekérdezéssel azt is le tudjuk kérdezni, hogy honnan hívtak meg egy adott függvényt oly módon, hogy az első paraméterének a lehetséges értékei között szerepel egy adott érték. Az alábbi lekérdezés a `mnesia_log` modul `open_log` függvényének keresi azon hívásait, ahol a lehetséges első paraméter érték éppen a `decision_log` atom:

$$mods[name = mnesia\_log].funs[name = open\_log].refs[.param[index = 1]$$

$$.origin[type = atom, value = decision\_log]]$$

Az *Origin* halmazokat és a kompakt adatfolyam reláció eredményeit számos magasabbrendű elemzés is felhasználja. Ilyen a dinamikus függvényhívás elemzés, mely az adatfolyam segítségével számolja a függvények/modulok lehetséges nevének értékeit és a paraméterlista hosszát [HK13]. Szükségünk van adatfolyam elemzésre az azonosítók lehetséges értékeinek a kiszámítására folyamatok [IM14, TB14, TB12a], szerverfolyamatok [BKT18, BT16], állapotok/állapotgépek [LTB18, LT18] és hierarchikus folyamatok modelljének a felderítésekor is [BST18].

Az elsőrendű adatfolyam reláció számításának hatékonysága kielégítőnek bizonyult a gyakorlati alkalmazásokban. Nagyméretű (több mint 1 millió soros adatbázisokon) is teszteltük a különböző alkalmazásain keresztül: a lekérdezőnyelvbe épített *.origin* lekérdezéssel, a dinamikus függvényhívás elemzés részeként stb.

Az alábbiakban ismertetek egy példa környezetet és eredményt az adatfolyam reláció valós használatának szemléltetésére:

- felépítettem a RefactorErl SPG-jét több mint négyszázezer sornyi[3] Erlang forráskódhoz;
- az elemzett programok az Erlang/OTP által nyújtott alkalmazások forráskódjából származtak (`megaco, mnesia, edoc, tools` stb;

[3]411714 sor, 33335 függvény definíció

- a dinamikus-függvényhívás elemzés 269 másodpercig tartott, melynek csak egy része az adatfolyam reláció számítása;
- 899 dinamikus-függvényhívás lett elemezve, mely során több mint 2500 kifejezés[4] *Origin* halmazát kellett kiszámolni a kompakt adatfolyam reláció segítségével;
- az adatfolyam számítások összesen 63 másodpercig tartottak különböző szűrésekkel együtt;
- 12 esetben a számítás nem végzett 10 másodpercen belül, ezért leállítottuk a számítást.

Ugyanezen az SPG-n elvégeztem további elemzéseket, mely során véletlenszerű kifejezéseket kiválasztva elsőrendű *Origin* halmazokat számítva azt kaptam, hogy az átlagos futási idő 0.005 és 0.3 másodperc között van az átlagot ezer számításonként számolva[5].

A számítások során a memóriahasználat 100 és 130 MB közötti skálán mozgott.

2.6. Kapcsolódó publikációk

A tézis eredményeit bemutató fő publikációk [TB12b, TBHT10]:

- Melinda Tóth and István Bozó. Static analysis of complex software systems implemented in Erlang. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 440–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

- Melinda Tóth, István Bozó, Zoltán Horváth, and Máté Tejfel. First order flow analysis for Erlang. In *8th Joint Conference on Mathematics*

[4]függvényhívásonként egy, kettő vagy három számítás szükséges
[5]Az összes számítás átlaga 0.08 másodperc volt.

and Computer Science, Selected papers, ISBN 978-963-9056-38-1, pages 403–416, Komárno, Slovakia, July 2010.

A tézis eredményeihez kapcsolódó további publikációk:

- állapotgépek elemzése: [LTB18, LT18]
- szoftver hierarchia elemzés: [BST18]
- szerver folyamatok elemzése: [BKT18, BT16]
- kommunikációs modell definiálása: [IM14, TB14, TB12a]
- párhuzamosítható minták felismerése és refaktorálása: [TBK17, BFH+15, BFH+14, KTBH16, KTB18]
- függőségi gráfok definiálása: [TB11, TBH+10b]
- hatáselemzés tesztesetszelekcióhoz: [HBT14, BT11, TBH13, FBT17]
- hatáselemzés refaktorálás validálására: [BTT+11a, BTT+11b]
- statikus elemzők validálása: [TTB+12]
- kódmegértés támogatása lekérdező nyelvvel: [TBH10a, HBKT11]
- párhuzamosítás elősegítése függőségek alapján: [HBTE10, TBHE11]

Az itt felsorolt publikációkra összesen 35 független hivatkozást jegyez az MTMT, ebből 8 független hivatkozás a kiemelt, eredményeket alátámasztó publikációkra történt.

3. fejezet

Adatfüggőség elemzés

Az előző fejezetben bemutatott adatfolyam elemzést arra is fel tudjuk használni, hogy eldönthessük azt, hogy egy adott adat megváltoztatása, milyen más egyéb program pontokon vezet az adatok megváltoztatáshoz. Ugyanis, ha tudjuk, hogy egy DFG-beli csúcsból egy másik csúcsba lemásolódhat az adat ($n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2$ valamely $n_1, n_2 \in N$ esetén), akkor ezen n_1 csúcshoz tartozó kifejezés értékének a megváltoztatása hatással van az n_2-re is. A szoftverek evolúcióját tekintve azonban nem csupán az adatfolyam reláció mentén terjedő változások jelentenek fenyegetést egy esetleges hibás működésre. Arra is figyelnünk kell, hogy egy részkifejezés megváltoztatása milyen más kifejezésekre lehet hatással a végrehajtás szempontjából. Az adatfolyam reláció a viselkedésre vett hatással együtt a szoftver különböző pontjain lévő kifejezései között is kapcsolatot teremthet. Ennek egy lehetséges alkalmazása lehet a releváns tesztesetek kiszűrése, mely során egy adott forráskód változáshoz szeretnénk megadni, hogy melyek azok a tesztesetek, amiket szükséges lefuttatni a helyes működés ellenőrzése érdekében.

Ezen kutatás a [LÖ09] cikkben megfogalmazott adatfolyam elemzés módosításán/kiegészítésén alapul. Az adatfolyam elemzés módosításának alapja az, hogy ha egy adat eljut egy n_1 csúcsból egy n_2 csúcsba, majd az n_2-ből az n_3-ba, akkor azt is mondhassuk, hogy az n_1-ből eljutott az n_3-ba. Azaz a reláció legyen tranzitív. A kifejezések függésének definíciója pedig erre a kiegészített nulladrendű adatfolyamra épül, melyet a 2.1. fejezetben mutattam be.

A B. függelékben részletezett motivációs példa rámutat (2. fejezet), hogy az

adatok változását követve ki tudunk szűrni változáshoz nem tartozó teszteseteket, és ezzel is rövidíteni tudjuk a tesztelésre fordított időt.

Most egy leegyszerűsített példán keresztül tekintsük a módszer szükségességét. Az `inc/1` függvény a paraméterül kapott számot megnöveli eggyel, a `static/0` függvény meghívja az `inc/1` függvényt és visszatér egy párral, majd a `test_/0` függvény[1] ellenőrzi annak visszatérési értékét.

```
inc(X) ->
    Ret = X + 1,
    Ret.

static() ->
    {0, inc(2)}.

test_() ->
    {0, 3} = static().
```

Ha átírjuk a `static/0` függvényünkben a pár első elemének, a `0`-nak, az értéket, akkor az adatfolyam reláció mentén a változás elterjed a `test_` függvénybe. Azonban, ha az `inc(2)` függvényhívásban írjuk át a `2` értéket, akkor az belefolyik az `X` változóba, de nem folyik át a függvényen a törzsében lévő infix művelet, az összeadás, miatt. Ugyanis az a művelet, függőségi élt generál az adatfolyam-gráfba. Ez a példa rámutat arra, hogy ha azt szeretnénk látni, hogy egy adattól milyen más értékek függnek, akkor egy újabb függőségi relációt is definiálnunk kell.

Az adatfolyam és -függőség információk mellett azonban szükségünk van arra is, hogy meg tudjuk mondani, hogy egy kifejezés értékének a megváltozása, hogyan hat a környezetére. Az adatfolyam-gráf építő szabályaihoz hasonlóan definiáltam kompozícionális szabályokat, melyek a nyelv különböző elemeire definiálják a részkifejezések közötti viselkedésbeli függőségeket. Ezekkel építjük fel az úgynevezett Viselkedésfüggőségi Gráfot (továbbiakban BDG, az angol Behaviour Dependence Graph kifejezésből). A $BDG = (N, E)$ egy olyan élcímkézett, irányított gráf, melynek csúcsai az Erlang szintaktikus részkifejezései, élei

[1] Az Erlang EUnit keretrendszerének megfelelően a test_ végű függvények a tesztek.

	Kifejezés	BDG élek
(Variable)	p egy változó kötési mintája n ezen változó használata	$p \xrightarrow{\mathbf{f}} n$
(List gen.)	e_0: $[e_1 \| \| p \leftarrow e_2]$	$e_1 \xrightarrow{\mathbf{c_e}} e_0$, $e_2 \xrightarrow{\mathbf{s_e}} p$ $e_1 \xrightarrow{\mathbf{b}} e_0$, $e_2 \xrightarrow{\mathbf{b}} e_0$

3.1. ábra. BDG építő szabályok

pedig a kifejezések közötti közvetett adatfolyamot, adatfüggőséget és viselkedést befolyásoló kapcsolatokat írják le. Azaz a BDG-t a DFG egy kiterjesztéseként kaphatjuk meg. A korábban ismertetett élekhez egy új éltípust vezetünk be: $n_1 \xrightarrow{\mathbf{b}} n_2$. Ennek jelentése, hogy n_2 viselkedése közvetlenül függ az n_1 viselkedésétől. Itt megjegyeznénk, hogy a korábban már bevezetett $\xrightarrow{\mathbf{d}}$, azaz függőségi élek $\xrightarrow{\mathbf{b}}$ élekként is viselkednek.

A 3.1. ábra mutatja az egyszerű változó használat és a lista értelmező kifejezés BGD építő szabályait. Az első példa azt mutatja, hogy a változó előfordulások és a változó kötési pontja között adat másolás kapcsolat van, de viselkedésbeli függés nincs. Azonban a másik példából láthatjuk, hogy a listaértelmező kifejezés függ a fejében lévő kifejezéstől. Ha például megváltozik a kifejezés oly módon, hogy egy hibát ad a kiértékelés közben, akkor az egész listaértelmező kifejezés is hibás lesz.

Egy másik példa a viselkedésbeli függésre a a függvények kiértékelése. A függvény őrfeltételétől függ a függvény törzsének kiértékelése, hiszen ha megváltoztatok az őrfeltételben egy adatot, akkor annak igazságértékét, ezzel pedig a kiértékelést is megváltoztatom.

3.1. Függőségi reláció

A BDG csak közvetlen adatfüggőségi éleket tartalmaz. Amennyiben a forráskód változás hatását le szeretnénk követni a teljes programra nézve, akkor a közvetett elérési információra is szükségünk van. Ehhez definiáljuk az $n_1 \rightsquigarrow n_2$ függőségi relációt. A reláció jelentése, hogy n_2 kiértékelése függ az n_1 értékétől/viselkedésétől. Tehát n_1 változása hatással lehet n_2-re.

A függőségi reláció definiálása két lépésben történik a korábban definiált

nulladrendű adatelérés reláció felhasználásával. Először definiáltam a viselkedés-függőségi relációt az alábbi minimális reláció segítségével[2]:

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2}{n_1 \overset{\mathbf{b}}{\rightsquigarrow} n_2} \qquad \text{(0f-rule)}$$

$$\frac{n_1 \overset{\mathbf{b}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{b}}{\rightarrow} n_3,\ n_3 \overset{\mathbf{b}}{\rightsquigarrow} n_4}{n_1 \overset{\mathbf{b}}{\rightsquigarrow} n_4} \qquad \text{(b-rule)}$$

Azaz az adatfolyam mentén és a $\overset{\mathbf{b}}{\rightarrow}$ élek mentén terjed a viselkedés függőség.

A függőségi relációt ($\rightsquigarrow$) pedig az alábbi tulajdonságokat teljesítő minimális relációként definiáljuk:

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2}{n_1 \rightsquigarrow n_2} \qquad \text{(data-rule)}$$

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{d}}{\rightarrow} n_3,\ n_3 \overset{\mathbf{b}}{\rightsquigarrow} n_4}{n_1 \rightsquigarrow n_4} \qquad \text{(b-dep-rule)}$$

A függőségi reláció részletes elemzése a B. függelékben található (4.2., 4.3., 4.4. fejezetek), és egy rövid összefoglalót ad az A függelékben szereplő első cikk 3.8. fejezete.

A tézist bemutató publikáció nulladrendű adatfolyamra épít, de a szabályokban a nulladrendű adatfolyam lecserélhető elsőrendűre is.

3.1.1. A *BDG* és a függőségi reláció bemutatása

A fejezet elején bemutattam egy motivációs példát, mellyel egy függőségi reláció szükségességére mutattam rá. Vizsgáljuk meg ezt a példát újra. Az lenne a cél, hogy az `inc(2)` függvényhívás paraméterének az értékének a változása/hatása elterjedhessen a `test_` függvénybe.

A 3.2. ábrán látható a motivációs kódrészlet, a 3.3. ábra pedig az ehhez tartozó *BDG* releváns részét mutatja be. Az ábrán a *BDG* csomópontok rendelkeznek egy egyedi azonosítóval és csomópont típussal (ez látható a csúcsok első sorában), melyek a RefactorErl szintaxisfájából származnak. A csúcsokat

[2] A 2.1. fejezetben $0f$ megnevezést használtam az adatelérés relációra, de a B. függelék d jelölést használ. A dolgozat jelölésrendszerének a konzisztenciája végett itt most $0f$-et írok.

```
inc(X) ->
    Ret = X + 1,
    Ret.

static() ->
    {0, inc(2)}.

test_() ->
    {0, 3} = static().
```

3.2. ábra. Szemléltető Erlang függvények

reprezentáló dobozok második sorában szerepel a kifejezés típusa és a hozzá tartozó szövegrészlet a forráskódból. Az ábra legtetején található `12`-es azonosítójú csúcs reprezentálja a forráskódban a `2` számot, mely az `inc(2)` függvényhívás aktuális paramétere. Az alábbiakban, ha a kifejezés csúcsának azonosítója i a gráfon, akkor a levezetésben e_i-ként fogok rá hivatkozni. Amennyiben egy mintakifejezésről van szó, akkor pedig p_i-ként.

A gráfon látható az, hogy az $e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} p_2$ és az $e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} e_4$ relációk fennállnak ugyan, de innen már sehová nem tudunk eljutni, így az e_{12} értéke nem folyik bele a `test_` függvénybe. Más szóval, az $Origin(e_{12}) = \{e_{12}, p_2, e_4\}$, és ezen halmaznak egyetlen eleme sem része a `test_` függvénynek.

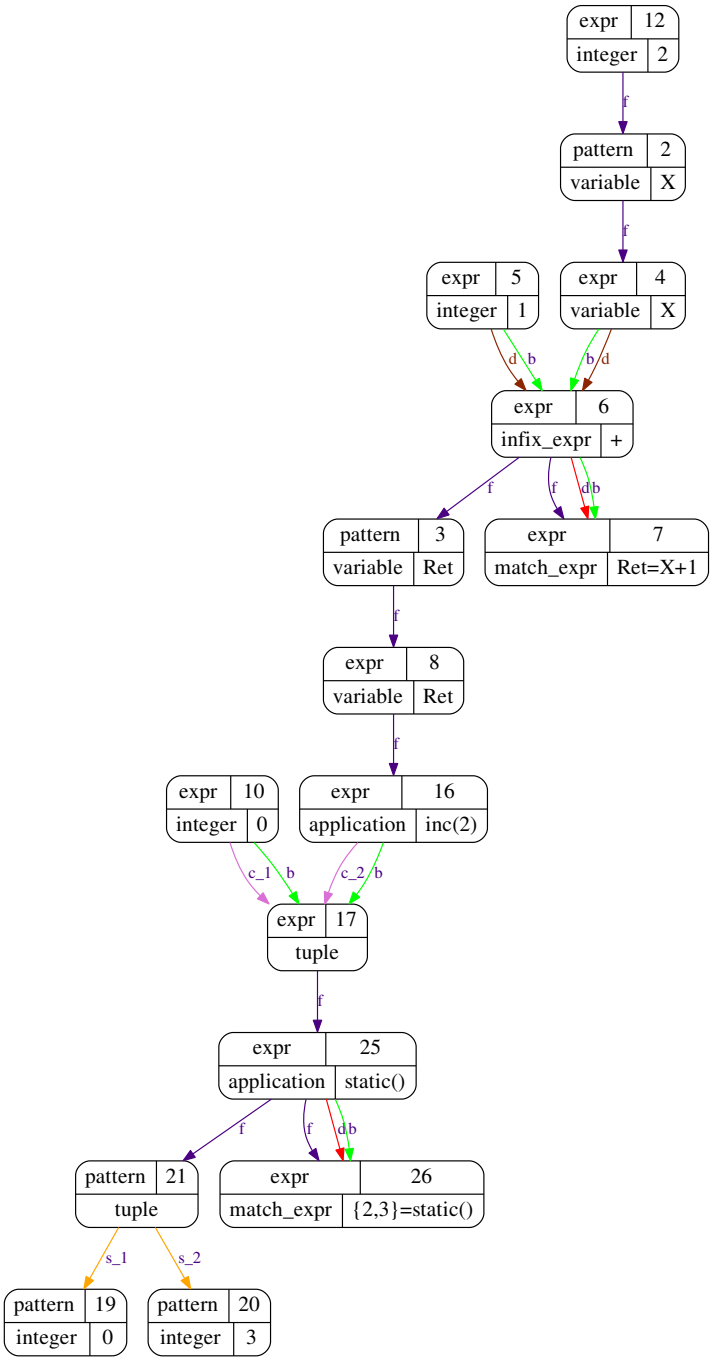
Azonban, ha nemcsak a $\overset{\mathbf{0f}}{\rightsquigarrow}$ relációt vesszük figyelembe, hanem a függéseket is, akkor az érték hatása elér a `test_` függvénybe. Tekintsük ennek igazolására az alábbi levezetést.

Az $e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} e_4$ reláció fennáll az f szabályok és a tranzitivitás miatt:

$$\frac{e_{12} \overset{\mathbf{f}}{\rightarrow} p_2,\ p_2 \overset{\mathbf{f}}{\rightarrow} e_4}{e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} p_2,\ p_2 \overset{\mathbf{0f}}{\rightsquigarrow} e_4} \qquad \text{(f rule (kétszer))}$$

$$\frac{e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} p_2,\ p_2 \overset{\mathbf{0f}}{\rightsquigarrow} e_4}{e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} e_4} \qquad \text{(tranzitivitás)}$$

Az $e_6 \overset{\mathbf{0f}}{\rightsquigarrow} e_{16}$ reláció fennáll az f szabályok és a tranzitivitás miatt:



3.3. ábra. Viselkedésfüggőségi gráf

$$\frac{e_6 \xrightarrow{\mathbf{f}} p_3,\ p_3 \xrightarrow{\mathbf{f}} e_8,\ e_8 \xrightarrow{\mathbf{f}} e_{16}}{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} p_3,\ p_3 \overset{\mathbf{0f}}{\rightsquigarrow} e_8,\ e_8 \overset{\mathbf{0f}}{\rightsquigarrow} e_{16}} \quad \text{(f rule (háromszor))}$$

$$\frac{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} p_3,\ p_3 \overset{\mathbf{0f}}{\rightsquigarrow} e_8,\ e_8 \overset{\mathbf{0f}}{\rightsquigarrow} e_{16}}{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} e_{16}} \quad \text{(tranzitivitás (kétszer))}$$

Az $e_{17} \overset{\mathbf{0f}}{\rightsquigarrow} p_{21}$ reláció fennáll az f szabályok és a tranzitivitás miatt:

$$\frac{e_{17} \xrightarrow{\mathbf{f}} e_{25},\ e_{25} \xrightarrow{\mathbf{f}} p_{21}}{e_{17} \overset{\mathbf{0f}}{\rightsquigarrow} e_{25},\ e_{25} \overset{\mathbf{0f}}{\rightsquigarrow} p_{21}} \quad \text{(f rule (kétszer))}$$

$$\frac{e_{17} \overset{\mathbf{0f}}{\rightsquigarrow} e_{25},\ e_{25} \overset{\mathbf{0f}}{\rightsquigarrow} p_{21}}{e_{25} \overset{\mathbf{0f}}{\rightsquigarrow} p_{21}} \quad \text{(tranzitivitás)}$$

Az $e_6 \overset{\mathbf{0f}}{\rightsquigarrow} p_{20}$ reláció fennáll a párba (be/ki)csomagolás és a tranzitivitás miatt:

$$\frac{e_{16} \xrightarrow{\mathbf{c_2}} e_{17},\ e_{17} \overset{\mathbf{0f}}{\rightsquigarrow} p_{21},\ p_{21} \xrightarrow{\mathbf{s_2}} p_{20}}{e_{16} \overset{\mathbf{0f}}{\rightsquigarrow} p_{20}} \quad \text{(c-s rule)}$$

$$\frac{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} e_{16},\ e_{16} \overset{\mathbf{0f}}{\rightsquigarrow} p_{20}}{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} p_{20}} \quad \text{(tranzitivitás)}$$

Mivel fennáll a nulladrendű adatelérés, ezért a viselkedési függőség is fennáll:

$$\frac{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} p_{20}}{e_6 \overset{\mathbf{b}}{\rightsquigarrow} p_{20}} \quad \text{(0f rule)}$$

A fentiek alapján $b - dep$ szabály alkalmazásával megkapjuk, hogy $e_{12} \rightsquigarrow p_{20}$:

$$\frac{e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} e_4,\ e_4 \xrightarrow{\mathbf{d}} e_6,\ e_6 \overset{\mathbf{b}}{\rightsquigarrow} p_{20}}{e_{12} \rightsquigarrow p_{20}} \quad \text{(b-dep rule)}$$

A $e_{12} \rightsquigarrow p_{20}$ relációból arra következtethetünk, hogy az e_{12} értékétől függ a `test_` függvény, mivel a p_{20} kifejezés a függvény része.

Megjegyezném, hogy az $e_6 \overset{\mathbf{b}}{\rightsquigarrow} p_{20}$ viselkedésfüggés igazolásához felhasználtuk a $c - s$ szabályt, majd a nulladrendű adatfolyamból következtettünk a visel-

kedésbeli függésre. Amennyiben nem lenne a gráfban $\overset{\mathbf{s_2}}{\rightarrow}$ él[3] és nem tudnánk alkalmazni a $c - s$ szabályt, akkor is tudunk a `test_` függvény érintettségére következtetni. Ugyanis e_{16} és e_{17} között van egy $\overset{\mathbf{b}}{\rightarrow}$ él is, mely segítségével levezethető az $e_6 \overset{\mathbf{b}}{\rightsquigarrow} e_{25}$ viselkedésfüggőség és így az $e_{12} \rightsquigarrow e_{25}$ függőség is:

$$\frac{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} e_{16}}{e_6 \overset{\mathbf{b}}{\rightsquigarrow} p_{16}} \quad \text{(0f rule)}$$

$$\frac{e_{17} \overset{\mathbf{f}}{\rightarrow} e_{25}}{e_{17} \overset{\mathbf{0f}}{\rightsquigarrow} p_{25}} \quad \text{(f rule)}$$

$$\frac{e_{17} \overset{\mathbf{0f}}{\rightsquigarrow} e_{25}}{e_{17} \overset{\mathbf{b}}{\rightsquigarrow} p_{25}} \quad \text{(0f rule)}$$

$$\frac{e_6 \overset{\mathbf{b}}{\rightsquigarrow} e_{16},\ e_{16} \overset{\mathbf{b}}{\rightarrow} e_{17},\ e_{17} \overset{\mathbf{b}}{\rightsquigarrow} e_{25}}{e_6 \overset{\mathbf{b}}{\rightsquigarrow} e_5} \quad \text{(b rule)}$$

$$\frac{e_{12} \overset{\mathbf{0f}}{\rightsquigarrow} e_4,\ e_4 \overset{\mathbf{d}}{\rightarrow} e_6,\ e_6 \overset{\mathbf{b}}{\rightsquigarrow} e_{25}}{e_{12} \rightsquigarrow e_{25}} \quad \text{(b-dep rule)}$$

Mivel e_{25} is része a `test_` függvénynek, ezért e_{12} változása érintheti a függvényt.

3.2. Kapcsolódó publikációk

A tézis eredményeit bemutató fő publikációk [TBH+10b, TB12b]:

- Melinda Tóth, István Bozó, Zoltán Horváth, László Lövei, Máté Tejfel, and Tamás Kozsik. Impact analysis of Erlang programs using behaviour dependency graphs. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, pages 372–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[3]Amennyiben lecseréljük a `{0,3} = static()` kifejezést `Res = static()` kifejezésre

- Melinda Tóth and István Bozó. Static analysis of complex software systems implemented in Erlang. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 440–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

A tézis eredményeihez kapcsolódó publikációk:

- párhuzamosítható minták felismerése és refaktorálása: [TBK17, BFH+15, BFH+14, KTBH16, KTB18]
- függőségi gráfok definiálása: [TB11]
- hatáselemzés tesztesetszelekcióhoz: [HBT14, BT11, TBH13, FBT17]
- hatáselemzés refaktorálás validálására: [BTT+11a, BTT+11b]
- párhuzamosítás elősegítése függőségek alapján: [HBTE10, TBHE11]

Az itt felsorolt publikációkra összesen 31 független hivatkozást jegyez az MTMT, ebből 10 független hivatkozás a kiemelt, eredményeket alátámasztó publikációkra történt.

4. fejezet

Párhuzamosítható minták felismerése

A szoftverfejlesztés és karbantartás során egy gyakori lépés a forráskód különböző szempontok szerinti transzformálása, refaktorálása [FBB+99]. Ezen szempont lehet a forráskód egységesítése, kódolási előírásokhoz való igazítása vagy éppen optimalizálása is. Egy optimalizálási szempont az, hogy az eddig szekvenciális kódrészleteket az új sokmagos számítógépes környezet adta lehetőségeket kihasználva hatékonyabban tudjuk futtatni, azaz párhuzamosítsunk. Ez a folyamat két lépésből áll. Egyrészt meg kell találnunk azokat a forráskód részleteket, jelölteket, amelyeket érdemes párhuzamosítani, másrészt végre kell hajtani a forráskód átírását. Kutatásunk célja a strukturált párhuzamosságot támogató `skel` könyvtár [Ske14, HAB+13] által támogatott minták bevezetése Erlang forráskódokba. Így azon jelöltek felismerésével foglalkoztunk, melynek párhuzamosítását a `skel` is támogatja.

Az alábbi párhuzamosítási minták felismerésével foglalkoztunk:

- task farm
- pipeline
- reduce
- divide and conquer, sort, search

- pool (feedback)

A felismerés során két csoportba osztottuk a felismerendő párhuzamosítható mintákat. Az első csoport (egyszerű jelöltek) foglalja magában azokat a nyelvi konstrukciókat és könyvtári hívásokat, melyek ezen jól ismert párhuzamosítható számítások szekvenciális modelljét írják le. A második csoport pedig (rekurzív jelöltek) a rekurzív függvények vizsgálatán alapul. Olyan rekurzív végrehajtási sémákat kerestünk adatfolyam, adatfüggőség, vezérlésfolyam és egyéb statikus szemantikus elemzésekre építve, melyek valamely párhuzamos programozási minta szekvenciális megfelelőjét írják le.

A mintafelismerés szabályait részletesen bemutatja a C. függelék. A függelék első cikkének az 5. fejezete a leginkább összefoglaló jellegű, de a második cikk 4.3. fejezete, a harmadik cikk 4. fejezete és a negyedik cikk 4.2 fejezete is bemutatja a felismerési szabályok egyes részeit. A fejezet további részében egy rövid összefoglalót adok a farm-jellegű (átalakítható párhuzamos task farm kódra) számítások felismeréséről. Farm-jellegűnek tekintjük az elemenkénti feldolgozást, amikor egy összetett adatszerkezet elemein egymástól függetlenül hajtunk végre egy számítást.

4.1. Egyszerű jelöltek keresése

A mintafelismerés ezen része adott tulajdonságú kifejezések keresését jelenti a forráskódban, ami legtöbbször a program szintaxisfájának bejárásával történik.

Az elemenkénti feldolgozás műveletét Erlangban kifejezés segítségével úgy írhatjuk le, ha listaértelmezőt használunk. Más iterációt leíró kifejezés listákhoz nem létezik. Az alábbi példa azt mutatja, hogyan tudjuk egy listaértelmező kifejezéssel kifejezni azt, hogy a lista összes elemére végre akarjuk hajtani a `foo/1` függvényt.

```
[foo(X) || X <- List]
```

A kiértékelés során a `List` lista minden elemére szekvenciálisan végrehajtjuk a `foo/1` műveletet, és a függvényhívások eredményét elhelyezzük egy új listába.

Szemantikájában a listaértelmezővel megegyező módon kifejezhetjük az elemenkénti feldolgozást a `lists:map/2` függvény segítségével is, ahol a végrehajtandó műveletet egy függvénykifejezés segítségével adjuk át paraméterül a könyvtári függvénynek.

```
lists:map(fun foo/1, List)
```

4.2. Rekurzív jelöltek keresése

A farm-jellegű párhuzamos számítások szekvenciális modellje leírható egy olyan rekurzív függvénnyel, mely valamely bemeneti adatstruktúra egy elemére elvégez egy műveletet, majd a számítást ettől az eredménytől függetlenül megismétli az adatszerkezet többi elemére is rekurzívan.

Tekintsük erre az alábbi példát, ahol a paraméterül kapott lista első eleméhez hozzáadjuk a `Value` változó értékét, majd rekurzívan meghívjuk az `inc/2` függvényt a lista farkára.

```
inc(Value,List) ->
  case List of
      [] -> [];
      [Head|Tail] ->
          X = Head + Value,
          [X | inc(Value,Tail)]
  end.
```

A lista elemein végzett számítások egymástól függetlenül, akár párhuzamosan is, elvégezhetőek. A függvény, amit a lista elemein párhuzamosan el kell végezni:

```
fun(E) -> E + Value end
```

Ennek a rekurzív függvénynek a `skel` könyvtár segítségével megadott párhuzamos átírása az alábbi:

```
inc(Value, List) ->
  Fun = fun(E) -> E + Value end,
  skel:do([{farm, {seq, Fun}, 10}], List).
```

Tekintsünk egy olyan példát, ahol ez a tulajdonság nem teljesül. A `foo/1` függvény törzsében a feldolgozás során úgy állítjuk elő a végeredménybe kerülő `X` változó értékét, hogy a bemeneti lista fej eleméhez hozzáadjuk a lista farkának a hosszát.

```
foo(List) ->
  case List of
      [] -> [];
      [Head|Tail] ->
          X = Head + length(Tail),
          [X | foo(Tail)]
  end.
```

Ez a példa azt mutatja, hogy a végeredményt nem tudjuk kiszámolni csak a lista egy elemének a függvényében, hanem szükségünk van arra az információra is, hogy éppen hol járunk a feldolgozásban, így ez a művelet nem párhuzamosítható.

Tekintsünk egy f függvényt, és a belőle épített adatfolyam- és vezérlésfolyam-gráfot. Az egyszerű lista adatszerkezeten dolgozó, nem végrekurzív függvényeket az alábbi esetben tekintjük farm-jellegűnek és ezzel együtt párhuzamosíthatónak:

1. f rekurzív: azaz a ICFG (Interprocedural Control Flow Graph) tartalmaz egy végrehajtási utat[1] f kezdeti pontjából ($start_f$) f egy hívási pontjába $call_f^c$. Az n csúcsból induló végrehajtási utak halmazát jelölje $EP(n)$:

$$\exists p \in EP(start_f),\ \exists c \text{ such that } call_f^c \in p$$

2. az f definíciójának van alapesete: létezik olyan irányított út a f kezdeti csúcsából a végső csúcsúba (end_f), melyen nem szerepel hívás f-re:

$$\exists p \in EP(start_f) \text{ such that } \left(\nexists c : call_f^c \in p\right) \wedge \left(end_f \in p\right)$$

3. f nem lehet divide-and-conquer jellegű, azaz nem hívhatja meg magát egynél többször ugyanazon a végrehajtási úton:

[1] egy irányított út az ICFG-ben

$$\forall c_1 \forall p \in EP(ret_f^{c_1}) : \left(\nexists c_2 : call_f^{c_2} \in p\right)$$

4. f-nek egy vagy több paramétere lehet, de ezek közül van egy kitüntetett, egy lista `L`, aminek az elemeit dolgozza fel a függvény: $f(\overline{V_1}, \texttt{L}, \overline{V_2})$. Definiáljuk `L`-hez az alábbi függvényeket:

$$Head(\texttt{L}) = \left\{ n \;\middle|\; \exists n' : \texttt{L} \overset{\mathbf{0f}}{\rightsquigarrow} n',\ n' \overset{\mathbf{s_h}}{\rightarrow} n \right\}$$
$$Tail(\texttt{L}) = \left\{ n \;\middle|\; \exists n' : \texttt{L} \overset{\mathbf{0f}}{\rightsquigarrow} n',\ n' \overset{\mathbf{s_t}}{\rightarrow} n \right\}$$

   Jelölje $\overset{\mathbf{s_h}}{\rightarrow}$ a lista fejére, $\overset{\mathbf{s_t}}{\rightarrow}$ pedig a lista farkába mutató adatfolyam éleket egy listás mintában.

5. További megkötéseket az f visszatérési értékeire teszünk, melyet R-ként definiálunk:

$$R = \left\{ \varrho \;\middle|\; \exists l :\ (\varrho, l, end_f) \in G_{CF}(f) \right\}$$

   R minden elemére az alábbiaknak kell teljesülni ($\varrho \in R$):

   (a) Ha ϱ nem rekurzív végrehajtási úton van, akkor üres listára kell kiértékelődnie:

$$\forall n\colon \quad n \overset{\mathbf{0f_c}}{\rightsquigarrow} \varrho \implies n = \texttt{[]}$$

   (b) Ha ϱ rekurzív végrehajtási út eleme, akkor egy nem üres listára kell kiértékelődnie, azaz az alábbi szintaktikus formák valamelyikére: `[`χ`|`τ`]` vagy `[`χ`]` `++` τ (ahol χ és τ tetszőleges kifejezések):

$$\forall n\colon \quad n \overset{\mathbf{0f_c}}{\rightsquigarrow} \varrho \implies \Big(\exists \chi, \tau\colon n = \texttt{[}\chi\texttt{|}\tau\texttt{]}\Big) \vee \Big(\exists \sigma, \tau\colon (n = \sigma\texttt{++}\tau) \wedge \big(\forall n'\colon n' \overset{\mathbf{0f_c}}{\rightsquigarrow} \sigma \implies \exists \chi\colon n' = \texttt{[}\chi\texttt{]}\big)\Big)$$

   (c) χ-re és τ-ra az alábbiaknak kell teljesülni még:

      i. χ csak $Head(\texttt{L})$ halmazbeli elemeken keresztül függhet `L`-től. Azaz minden $\texttt{L} \overset{\mathbf{dep}}{\rightsquigarrow} \chi$ reláció esetén létezik $n \in Head(\texttt{L})$ úgy,

```
count_funstats(_,[]) ->
    [];
count_funstats(Agents,
               [{Stat, MapFun, ReduceFun, OldAcc} | T]) ->
    NewAcc =  lists:foldl(ReduceFun,
                          OldAcc,
                          [MapFun(Agent) || Agent <- Agents]),
    [{Stat, MapFun, ReduceFun, NewAcc} |
                          count_funstats(Agents,T)].
```

4.1. ábra. Farm-jellegű függvény

hogy $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} n$ és $n \overset{\mathbf{dep}}{\rightsquigarrow} \chi$.

ii. τ az f egy rekurzív hívásának eredményével egyezik meg, mely hívás paraméterei $\overline{V_1}$, az L lista farka és $\overline{V_2}$, azaz: $\exists n, \forall n'$:

$$n = f(\overline{\nu_1}, \alpha, \overline{\nu_2}) \ \wedge \ \overline{V_i} \overset{\mathbf{0f}}{\rightsquigarrow} \overline{\nu_i} \ (i = 1, 2) \ \wedge$$

$$n' \in \mathit{Tail}(\mathtt{L}) \ \wedge \ n' \overset{\mathbf{0f}}{\rightsquigarrow} \alpha \ \wedge \ n \overset{\mathbf{0f}}{\rightsquigarrow} \tau.$$

Amennyiben a függvény végrekurzív, vagy nem állít elő listát outputként, vagy éppen nem listákon, hanem konkurens módon érkező üzenetsorokat dolgozz fel, más-más szabályhalmazt definiáltunk a felismeréshez. Ezeknek természetesen gyakran van egy közös magja. Ezek részletes leírása a C. függelékben található (részletesen az első cikk 5.4. és 5.5. fejezeteiben).

4.2.1. Megtalált jelölt

A 4.1. ábrán látható kódrészlet a mintafelismerés által talált jelölt egy Erlang multi ágens rendszer szoftverében[2].

A farm jellegű számításokra megfogalmazott tulajdonságok teljesülnek rá:

- A `count_funststs/2` függvény rekurzív, van alapesete és nem tartalmaz többszörösen rekurzív végrehajtási utat egy függvénytörzs kiértékelésén belül (1.-3. tulajdonság).

[2]MAS – https://github.com/ParaPhraseAGH/erlang

```
count_funstats(Agents, List) ->
  Fun =
    fun({Stat, MapFun, ReduceFun, OldAcc}) ->
      NewAcc = lists:foldl(ReduceFun,
                           OldAcc,
                           [MapFun(Agent) || Agent <- Agents]),
      {Stat, MapFun, ReduceFun, NewAcc}
    end,
  skel:do([{farm, {seq, Fun}, 10}], List).
```

4.2. ábra. Farm-jellegű függvény a `skel` könyvtárra átírva

- A függvénynek van egy kitüntetett lista paramétere (4. tulajdonság) és egy extra argumentuma, az `Agents`. Ezt a változót a rekurzív hívásban változatlanul továbbítja a függvény a lista farkával együtt (5.c.ii. tulajdonság): τ = `count_funstats(Agents,T)`.

- A nem rekurzív végrehajtási úton a függvény üres listára (`[]`) értékelődik ki (5.a. tulajdonság). A rekurzív ágon pedig egy lista konstrukció lesz a visszatérési érték (5.b. tulajdonság):
  `[{Stat,MapFun,ReduceFun,NewAcc} | count_funstats(Agents,T)]`

- Az eredményül visszaadott listába beszúrt elem (χ) csak az eredeti lista fejelemétől (`{Stat, MapFun, ReduceFun, OldAcc}`) és a nem lista paramétertől (`Agents`) függ (5.c.i. tulajdonság):
  χ = `{Stat,MapFun,ReduceFun,NewAcc}`

A `count_funstats/2` függvény egy lehetséges átírását mutatja a 4.2. ábra. A `Fun` nevű változóba kiemelt függvény, az úgynevezett kernelfüggvény, jól látható módon nem függ a a lista farkától, csak a fejelemétől.

A mintafelismerést nyílt forráskódú szoftvereken teszteltük. Ennek eredményét mutatja be a C. függelék első cikkének 6. fejezete.

4.3. Kapcsolódó publikációk

A tézis eredményeit bemutató publikációk [TBK17, BFH+15, BFH+14, KTBH16]:

- Melinda Tóth, István Bozó, and Tamás Kozsik. Pattern Candidate Discovery and Parallelization Techniques. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages*, 27 pages, New York, NY, USA, 2017. ACM.

- Tamás Kozsik, Melinda Tóth, István Bozó, and Zoltán Horváth. Static analysis for divide-and-conquer pattern discovery. *Computing and Informatics*, 35:764–791, 01 2016.

- István Bozó, Viktória Fördős, Dániel Horpácsi, Zoltán Horváth, Tamás Kozsik, Judit Kőszegi, and Melinda Tóth. Refactorings to enable parallelization. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming*, pages 104–121, Cham, 2015. Springer International Publishing.

- István Bozó, Viktória Fördős, Zoltán Horváth, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. Discovering parallel pattern candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, New York, NY, USA, 2014. ACM.

A tézis eredményeihez kapcsolódó publikációk:

- divide-and-conquer függvények refaktorálása: [KTB18]

- adatfolyam, adatfüggőség, vezérlésfolyam: [TBH$^+$10b, TB12b, TBHT10]

Az itt felsorolt publikációkra összesen 28 független hivatkozást jegyez az MTMT, ebből 14 független hivatkozás a kiemelt, eredményeket alátámasztó publikációkra történt.

5. fejezet

Összegzés

A szoftverfejlesztést támogató eszközök jelentősége rohamosan nő az utóbbi évtizedekben. A forráskódok mérete akkorára nő, hogy humán erővel átlátni szinte lehetetlen, de legalábbis nehézkes és időigényes folyamat. Így egyre inkább elterjednek azok az eszközök, melyek a kód megértést, karbantartást, hibakeresést támogatnak vagy éppen lehetőséget nyújtanak a forráskód különböző szempontok szerinti refaktorálására. Ez a támogatás történhet dinamikusan, azaz futási időben, illetve statikusan, azaz fordítási időben. Előbbi esetben a már futó szoftver monitorozásával, esetlegesen a kód instrumentálásával nyerhetünk ki információt és segíthetjük ezzel a fejlesztőket. Utóbbi esetben viszont nincs szükség a szoftver futtatására, csupán a forráskód alapján gyűjtünk információt és használjuk fel különböző célokra. Dolgozatomban ez utóbbi módszerrel, a forráskódok statikus elemzésével foglalkoztam, Erlang nyelvhez.

Definiáltam Erlang programok elsőrendű adatfolyam-gráfját, és az ezen a gráfon értelmezett elsőrendű adatfolyam relációt. A RefactorErl keretrendszer alap eszközkészletéhez igazítva megadtam az ehhez tartozó algoritmusokat. Erlang programok adatfolyam-gráfját így inkrementálisan fel tudjuk építeni a forráskódok változásának figyelembe vételével. A adatfolyam reláció segítségével pedig választ kaphatunk olyan kérdésekre, hogy mi lehet a program egy adott pontján lévő kifejezésének az értéke, illetve hogy egy adott érték milyen programpontokra juthat el. A reláció interprocedurálisan számítható és figyelembe veszi a hívási kontextust. Az adatfolyam relációt felhasználásával

megadtam, hogyan számítható ki az aszinkron üzenetküldések és -fogadások közötti közvetlen adatfolyam.

Definiáltam Erlang programokon értelmezett adatfüggőség relációt, mely az adatfolyam-gráf kiterjesztésén a viselkedésfüggőség gráfon számítható ki. A függőségi reláció megadja, hogy két Erlang programbeli kifejezés között van-e függés, azaz kiértékelésük függ-e egymástól. A RefactorErl keretrendszerhez kidolgozott algoritmusok felhasználásra kerültek olyan problémák megoldásában mint a releváns tesztesek kiválasztása, vagy párhuzamosítható komponensek függőségi kapcsolatainak az ellenőrzése.

Az adatfolyam, adatfüggőség és egyéb statikus elemzések felhasználásával kidolgoztam különböző jól párhuzamosítható számítási modellek viselkedésének leírását, mint például az elemenkénti feldolgozás. A megadott szabályok alapján a RefactorErl keretrendszerben megadhatók azok a mintafelismerési algoritmusok, melyek segítségével azonosíthatóak azok a szekvenciális kódrészletek, melyek lecserélhetőek egy ekvivalens párhuzamos végrehajtásra.

6. fejezet

Summary

The importance of the tools to support software development is increasing. The size of the software products makes manual scanning for certain information almost impossible, or time consuming. Therefore tools to support code comprehension, development, maintenance, debugging, or even automatic source code transformation are really desired. We can distinguish dynamic and static tools. The former analyses the software at runtime by monitoring, instrumenting the code. The latter analyses the source code itself without executing the program. In my thesis I have developed new static analyses methods for the Erlang programming language to support code comprehension and further static analyses.

I have defined the first order data flow graph for Erlang programs, and the data flow relation among the nodes of the graph, the first order data flow reaching. The reaching relation itself is able to identify the possible values of an expression at some point in the program. It also identifies the expressions where a certain value may flow. Based on the definitions I have introduced the data flow graph building and reaching calculation algorithms using the RefactorErl framework. The data flow graph building algorithm is incremental, therefore the changes of the source code can be handled without reanalysing the whole software. The data flow reaching algorithm is interprocedural and aware of function call context. Using data flow reaching I have defined the direct data flow among asynchronous message sending and receiving expressions.

I have defined a data dependence relation among Erlang expressions based on the behaviour dependence graph that is an extension of the data flow graph. The dependence relation defines whether two expressions from the Erlang source code depends on each other. The corresponding algorithms have been defined in the framework of RefactorErl, and have been used in further static analyses, namely in change impact analysis and pattern discovery.

I have defined the behaviour of parallelisable computations, such as the elementwise processing. The definitions use the studied data flow, data dependence relations and other static analysis methods, such as control flow graphs and execution paths. Using the definitions we can identify sequential code fragments that can be replaced with parallel equivalents. The pattern discovery algorithms have been defined in the RefactorErl framework.

Hivatkozott művek listája

[Bin07] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[Boz18] István Bozó. Funkcionáli programok statikus elemzése és szeletelése, 2018. Doktori dolgozat.

[Car01] Richard Carlsson. An introduction to Core Erlang. In *In Proceedings of the PLI'01 Erlang Workshop*, 2001.

[CLPJ+07] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[Cpp18] Cppcheck. A tool for static c/c++ code analysis. http://cppcheck.sourceforge.net/, 2018. Accessed: 2018-07-23.

[CS11a] Maria Christakis and Konstantinos Sagonas. Detection of asynchronous message passing errors using static analysis. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages*, pages 5–18, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[CS11b] Maria Christakis and Konstantinos Sagonas. Static detection of deadlocks in erlang. Technical report, 2011.

[CT09] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly Media, 2009.

[EN08] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Gra18] GrammaTech. Codesonar. https://www.grammatech.com/products/codesonar, 2018. Accessed: 2018-07-23.

[GS15] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi, editors, *Intelligent Computing and Applications*, pages 581–591, New Delhi, 2015. Springer India.

[HAB+13] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *Formal Methods for Components and Objects*, volume 7542 of *LNCS*, pages 218–236. Springer Berlin Heidelberg, 2013.

[HK13] Dániel Horpácsi and Judit Koszegi. Static analysis of function calls in erlang. refining the static function call graph with dynamic call information by using data-flow analysis. *e-Informatica Software Engineering Journal*, 7(1), 2013.

[HSHZ09] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. Profiling Java Programs for Parallelism. In *Proc. IWMSE '09*, pages 49–55, 2009.

[KA10] Radoslav Kirkov and Gennady Agre. Source code analysis–an overview. *Cybernetics and Information Technologies*, 10(2):60–77, 2010.

[KOHB15] Dániel KRUPP, György ORBÁN, Gábor HORVÁTH, and Bence BABATI. Industrial experiences with the clang static analysis toolset. Presentations at 2015Euro LLVM Conference, 2015.

[LÖ9] László Lövei. Automated module interface upgrade. In *Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG*, ERLANG '09, pages 11–22, New York, NY, USA, 2009. ACM.

[LMC10] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[LS04] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Wei-Ngan Chin, editor, *Programming Languages and Systems*, pages 91–106, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[LS06] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 167–178. ACM, 2006.

[LT08] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 199–203, New York, NY, USA, 2008. ACM.

[LT11] Huiqing Li and Simon Thompson. Incremental clone detection and elimination for erlang programs. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software*

Engineering, pages 356–370, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[LT15] Huiqing Li and Simon Thompson. Safe concurrency introduction through slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 103–113, New York, NY, USA, 2015. ACM.

[MIK97] Greg Michaelson, Andrew Ireland, and Peter King. Towards a Skeleton Based Parallelising Compiler for SML. In *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, 1997.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

[NNH05] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999, corrected 2005.

[Ora18] Oracle. Opengrok. http://oracle.github.io/opengrok/, 2018. Accessed: 2018-07-23.

[PBKC18] Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 361–369, New York, NY, USA, 2018. ACM.

[RBS13] Gordana Rakic, Zoran Budimac, and Milos Savic. Language independent framework for static code analysis. In *Proceedings of the Balkan Conference in Informatics, BCI '13, Thessaloniki, Greece, September 19-21, 2013*, pages 236–243. ACM, 2013.

[SCI18] SCItools. Understand. https://scitools.com/, 2018. Accessed: 2018-07-23.

[Ske14] Skel Tutorial. Available at `http://chrisb.host.cs.st-andrews.ac.uk/skel-test-master/tutorial/bin/tutorial.html`, 2014.

[STT12] Josep Silva, Salvador Tamarit, and César Tomás. System dependence graphs in sequential erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 486–500, Berlin, Heidelberg, 2012. Springer-Verlag.

[SV12] Sudhakar Sah and Vinay G Vaidya. A review of parallelization tools and introduction to easypar. *International Journal of Computer Applications*, 56(12), 2012.

[Wög05] Wolfgang Wögerer. A survey of static program analysis techniques. Technical report, 2005.

Publikációs lista 1. – Tézisekhez

[BFH+14] István Bozó, Viktória Fördős, Zoltán Horváth, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. Discovering parallel pattern candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, New York, NY, USA, 2014. ACM.

[BFH+15] István Bozó, Viktória Fördős, Dániel Horpácsi, Zoltán Horváth, Tamás Kozsik, Judit Kőszegi, and Melinda Tóth. Refactorings to enable parallelization. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming*, pages 104–121, Cham, 2015. Springer International Publishing.

[BKT18] István Bozó, Mátyás Béla Kuti, and Melinda Tóth. Analysing and visualising callback modules of Erlang generic server behaviours. In *Proceedings of the 11th Joint Conference on Mathematics and Computer Science, CEUR Workshop Proceedings*, volume 2046, pages 23–41, 2018.

[BST18] István Bozó, Bence János Szabó, and Melinda Tóth. Analysing the hierarchical structure of Erlang applications. In *Proceedings of the 11th Joint Conference on Mathematics and Computer Science, CEUR Workshop Proceedings*, volume 2046, pages 42–55, 2018.

[BT11] István Bozó and Melinda Tóth. Selecting Erlang test cases using impact analysis. In Simos BE, editor, *International Conference on Numerical Analysis and Applied Mathematics: ICNAAM 2011,*

AIP Conference Proceedings, 1389, pages 802–805, Melville (NY), 2011. American Institute of Physics.

[BT16] István Bozó and Melinda Tóth. Analysing and visualising Erlang behaviours. In Theodore Simos and Charalambos Tsitoura, editors, *International Conference on Numerical Analysis and Applied Mathematics: 2015 ICNAAM, AIP Conference Proceedings, 1738*, Melville (NY), 2016. AIP Publishing. Art. No.: 240004.

[BTT+11a] István Bozó, Melinda Tóth, Máté Tejfel, Dániel Horpácsi, Róbert Kitlei, Judit Kőszegi, and Zoltán Horváth. Using impact analysis based knowledge for validating refactoring steps. In Militon Frentiu, Horia F. Pop, and Simona Motogna, editors, *International Conference on Knowledge Engineering, Principles and Techniques (KEPT 2011): Selected papers, 407 p.*, pages 325–336, Cluj-Napoca, 2011. Presa Universitara Clujeana.

[BTT+11b] István Bozó, Melinda Tóth, Máté Tejfel, Dániel Horpácsi, Róbert Kitlei, Judit Kőszegi, and Zoltán Horváth. Using impact analysis based knowledge for validating refactoring steps. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 56(3):57–64, 2011.

[FBT17] Viktória Fördős, István Bozó, and Melinda Tóth. Towards change-driven testing. In Natalia Chechina and Scott Lystig Fritchie, editors, *Erlang 2017: Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang*, pages 64–65, New York, 2017. ACM Press.

[HBKT11] Zoltán Horváth, István Bozó, Judit Kőszegi, and Melinda Tóth. Static analysis based support for program comprehension in Erlang. *ACTA ELECTROTECHNICA ET INFORMATICA*, 3:3–10, 2011.

[HBT14] István Bozó, Melinda Tóth and Zoltán Horváth. Reduction of regression tests for Erlang based on impact analysis. *STUDIA UNIVER-*

SITATIS BABES-BOLYAI SERIES INFORMATICA, 59(Special Issue 1):31–46, 2014.

[HBTE10] Zoltán Horváth, István Bozó, Melinda Tóth, and Attila Erdődi. Dependency graphs for parallelizing Erlang programs. In Hage Jurriaan, editor, *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages: IFL 2010, 423 p.*, pages 180–186, Utrecht, 2010. Utrecht University.

[HNL+08] Zoltán Horváth, Tamás Nagy, László Lövei, Melinda Tóth, and Anikó Nagyné Víg. Call graph and data flow analysis of a dynamic functional language. In *Proceedings of the Sixth Conference of PhD Students in Computer Science (CSCS'08), Extended abstract*, pages 42–43, Szeged, Hungary, 2008.

[IM14] Bozó István and Tóth Melinda. Erlang folyamatok és a köztük lévő kapcsolatok elemzése. In Csörnyei Zoltán, editor, *Tízéves az ELTE Eötvös József Collegium Informatikai Műhelye: 2004-2014*, pages 79–92, Budapest, 2014. ELTE Eötvös József Collegium.

[KTB18] Tamás Kozsik, Melinda Tóth, and István Bozó. Free the conqueror! refactoring divide-and-conquer functions. *Future Generation Computer Systems*, 79:687 – 699, 2018.

[KTBH16] Tamás Kozsik, Melinda Tóth, István Bozó, and Zoltán Horváth. Static analysis for divide-and-conquer pattern discovery. *Computing and Informatics*, 35:764–791, 01 2016.

[LT18] Dániel Lukács and Melinda Tóth. Translating Erlang state machines to uml using triple graph grammars. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 63(1):33–50, 2018.

[LTB18] Dániel Lukács, Melinda Tóth, and István Bozó. Transforming Erlang finite state machines. In *Proceedings of the 11th Joint Conference on Mathematics and Computer Science, CEUR Workshop Proceedings*, volume 2046, pages 197–218, 2018.

[TB11] Melinda Tóth and István Bozó. Building dependency graph for slicing Erlang programs. *PERIODICA POLYTECHNICA-ELECTRICAL ENGINEERING*, 55(3-4):133–138, 2011.

[TB12a] Melinda Tóth and István Bozó. Detecting process relationships in Erlang programs. In Ralf Hinze, editor, *Draft Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, pages 494–508, Oxford, 2012.

[TB12b] Melinda Tóth and István Bozó. Static analysis of complex software systems implemented in Erlang. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 440–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[TB14] Melinda Tóth and István Bozó. Detecting and visualising process relationships in Erlang. *PROCEDIA COMPUTER SCIENCE*, 29:1524–1534, 2014.

[TBH10a] Melinda Tóth, István Bozó, and Zoltán Horváth. Applying the query language to support program comprehension. In *Proceedings of International Scientific Conference on Computer Science and Engineering*, pages 52–59, Stara Lubovna, Slovakia, 2010.

[TBH+10b] Melinda Tóth, István Bozó, Zoltán Horváth, László Lövei, Máté Tejfel, and Tamás Kozsik. Impact analysis of Erlang programs using behaviour dependency graphs. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, pages 372–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[TBH13] Melinda Tóth, István Bozó, and Zoltán Horváth. Reduction of regression tests for Erlang based on impact analysis. In Peter Achten,

editor, *Draft Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, pages 1–7, 2013.

[TBHE11] Melinda Tóth, István Bozó, Zoltán Horváth, and Attila Erdődi. Static analysis and refactoring towards Erlang multicore programming. In Vivek Sarkar and Vasco T. Vasconcelos, editors, *Pre-Proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES'11*, pages 43–50, Saarbrücken, Germany, 2011.

[TBHT10] Melinda Tóth, István Bozó, Zoltán Horváth, and Máté Tejfel. First order flow analysis for Erlang. In *8th Joint Conference on Mathematics and Computer Science, Selected papers, ISBN 978-963-9056-38-1*, pages 403–416, Komárno, Slovakia, July 2010.

[TBK17] Melinda Tóth, István Bozó, and Tamás Kozsik. Pattern Candidate Discovery and Parallelization Techniques. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages*, p. 1-26, New York, NY, USA, 2017. ACM Press.

[TTB+12] Máté Tejfel, Melinda Tóth, István Bozó, Dániel Horpácsi, and Zoltán Horváth. Improving quality of software analyser and transformer tools using specification based testing. *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE SECTIO COMPUTATORICA*, 37:355–368, 2012.

Publikációs lista 2. – Egyéb

[ATAB16] Tamás Ambrus, Melinda Tóth, Domonkos Asztalos, and Zsófia Borbély. Tool to measure and refactor complex UML models. In Zoran Budimac, Zoltán Horváth, and Tamás Kozsik, editors, *Proceedings of the Fifth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA 2016), CEUR Workshop Proceedings*, volume 1677, 2016. Art. No.: 1, 8p.

[BHH+11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Köszegi, Máté Tejfel, and Melinda Tóth. RefactorErl - source code analysis and refactoring in Erlang. In Jaan Phenjam, editor, *Proceedings of the 12th Symposium on Programming Languages and Software Tools, SPLST 2011*, pages 138–148, Tallinn, 2011. TUT Press.

[DGM+16] Gergely Dévay, Tibor Gregorics, Tóth Melinda, Domonkos Asztalos, Dávid János Németh, Gábor Ferenc Kovács, Zoltán Gera András Dobreff Balázs Gregorics András Nagy Boldizsár Németh, Martin Budai, Zsolt Kulik, and Kristóf Kanyó. txtUML. In Juan de Lara, Peter J. Clarke, and Mehrdad Sabetzadeh, editors, *Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), CEUR Workshop Proceedings*, volume 1725, pages 8–15, 2016.

[FT13] Viktória Fördős and Melinda Tóth. Identifying code clones with RefactorErl. In Kiss Ákos, editor, *Proceedings of the 13th Sympo-*

sium on Programming Languages and Software Tools, SPLST'13, pages 31–45, Szeged, 2013. University of Szeged.

[FT14] Viktória Fördős and Melinda Tóth. Utilising the software metrics of RefactorErl to identify code clones in Erlang. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 59(Special Issue 1.):103–118, 2014.

[FT15a] Viktória Fördős and Melinda Tóth. Comprehensible presentation of clone detection results. In Theodore Simos and Charalambos Tsitouras, editors, *Proceedings of the International Conference on Numerical Analysis and Applied Mathematics 2014 (ICNAAM-2014): 4th Symposium on Computer Languages, Implementations and Tools, AIP Proceedings*, volume 1648. AIP Publishing, 2015. Art. No.: 310006, 4 p.

[FT15b] Viktória Fördős and Melinda Tóth. Supporting comprehensible presentation of clone candidates through two-dimensional maximisation. *COMPUTER LANGUAGES SYSTEMS & STRUCTURES*, 44(Part C):355–365, 2015.

[FT16] Viktória Fördős and Melinda Tóth. Identifying code clones with RefactorErl. *ACTA CYBERNETICA*, 22(3):553–571, 2016.

[FTK14] Viktória Fördős, Melinda Tóth, and Tamás Kozsik. Clone wars. In Zoran Budimac and Tihana Galinac Grbac, editors, *Proceedings of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA 2014), CEUR Workshop Proceedings*, volume 1266, pages 15–22, 2014.

[HLK+08] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Building a refactoring tool for Erlang. In K. Mens, M. van den Brand, A. Kuhn, H. M. Kienle, and R. Wuyts, editors, *Proceedings of the 1st International Workshop on Academic Software*

Development Tools and Techniques (WASDeTT-1), 2008. Art. No.: Submissions 12, 11 pages.

[HLK+09a] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, István Bozó, Melinda Tóth, and Roland Király. Modeling semantic knowledge in Erlang for refactoring. In *International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Selected papers*, pages 38–53, 2009. Invited paper. Earlier version appeared in Studia Universitatis Informatica, 54:7-16, 2009.

[HLK+09b] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling semantic knowledge in Erlang for refactoring. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 54:7–16, 2009. Special issue for Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Cluj-Napoca (Romania), Volume I.

[HT10] Zoltán Horváth and Melinda Tóth. RefactorErl: a source code analyser and transformer tool. In *Proceedings of the Erlang User Conference*, pages 24–26, 2010.

[HTL10] Lilla Hajós, Melinda Tóth, and László Lövei. Erlang Semantic Query Language. In Attila Egri-Nagy, Emod Kovács, Gergely Kovásznai, Gábor Kusper, and Tibor Tómács, editors, *Proceedings of the 8th International Conference on Applied Informatics*, pages 165–172, Eger, 2010. Eszterházy Károly College.

[KBK+10] Róbert Kitlei, István Bozó, Tamás Kozsik, Máté Tejfel, and Melinda Tóth. Analysis of preprocessor constructs in Erlang. In Scott Lystig Fritchie and Konstantinos F. Sagonas, editors, *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, pages 45–55, New York, 2010. ACM Press.

[KBT18] Mátyás Komáromi, István Bozó, and Melinda Tóth. An efficient graph visualisation framework for RefactorErl. *STUDIA UNIVER-*

SITATIS BABES-BOLYAI SERIES INFORMATICA, 63(2):21–35, 2018.

[KCH+08] Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg. Use cases for refactoring in erlang. In *Central European Functional Programming School: Second Summer School, CEFP 2007, Cluj-Napoca, Romania, June 23-30, 2007, Revised Selected Lectures*, volume 5161, pages 250–285, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[KLJ+13] Tamás Kozsik, András Lorincz, Dávid Juhász, László Domoszlai, Dániel Horpácsi, Melinda Tóth, and Zoltán Horváth. Workflow description in cyber-physical systems. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 58(2):20–30, 2013.

[KLT+08] Róbert Kitlei, László Lövei, Melinda Tóth, Zoltán Horváth, Tamás Kozsik, Roland Király, István Bozó, Csaba Hoch, and Dániel Horpácsi. Automated syntax manipulation in refactorerl. In *Proceedings of 14th International Erlang/OTP User Conference*, 2008.

[KT11] Mátyás Karácsonyi and Melinda Tóth. Analysing Erlang BEAM files. In Valerie Novitzká and Stefan Hudák, editors, *Proceedings of the 11th International Conference on Informatics 2011*, pages 90–95, Kassa, 2011. Technical University of Kosice Faculty of Electrical Engineering and Informatics.

[LT17] Dániel Lukács and Melinda Tóth. Structuring Erlang BEAM control flow. In Natalia Chechina and Scott Lystig Fritchie, editors, *Erlang 2017: Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang*, pages 31–42, New York, 2017. ACM Press.

[LTOT08] Huiqing Li, Simon Thompson, György Orosz, and Melinda Tóth. Refactoring with Wrangler, updated: data and process refactorings, and integration with Eclipse. In Horváth Zoltán, editor, *Erlang '08,*

Proceedings of the SIGPLAN Erlang Workshop, pages 61–72, New York, 2008. ACM Press.

[MLT18] Gregory Morse, Dániel Lukács, and Melinda Tóth. Incremental decompilation of loop-free binary code: Erlang. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 63(2):66–87, 2018.

[MNBT18] Áron Attila Mészáros, Gergely Nagy, István Bozó, and Melinda Tóth. Towards green computing in Erlang. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 63(1):64–79, 2018.

[NT14] András Németh and Melinda Tóth. Erlang-like dynamic typing in C++. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 59(Special Issue 1):185–200, 2014.

[OHKT14] Gábor Oláh, Dániel Horpácsi, Tamás Kozsik, and Melinda Tóth. Type inference for core Erlang to support test data generation. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 59(Special Issue 1):201–215, 2014.

[PBK+18] Artúr Poór, István Bozó, Tamás Kozsik, Gábor Páli, and Melinda Tóth. Benefits of implementing a query language in purely functional style. In *Proceedings of the 11th Joint Conference on Mathematics and Computer Science, CEUR Workshop Proceedings*, volume 2046, pages 250–266, 2018.

[PKTB18] Artúr Poor, Tamás Kozsik, Melinda Tóth, and István Bozó. Compiler front end fusion: Undo desugaring in language processing tools. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 63(2):5–20, 2018.

[RKF+18] Anna Reale, Péter Kiss, Charles Ferrari, Benedek Kovács, László Szilágyi, and Melinda Tóth. Application functions placement

optimization in a mobile distributed cloud environment. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 63(2):37–52, 2018.

[RTH17] Anna Reale, Melinda Tóth, and Zoltán Horváth. Towards context aware computations offloading in 5G. In Daniel-Jesus Munoz, Mónica Pintoó, and Lidia Fuentes, editors, *Proceeding ECSA '17 Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pages 89–92, New York, 2017. ACM Press.

[TB09] Melinda Tóth and István Bozó. Restructuring Erlang programs using function related refactorings. In Peltonen Jar, editor, *Proceedings of the 11th Symposium on Programming Languages and Software Tools SPLST'09*, pages 162–176, Helsinki, 2009. Nordic Academic Press.

[TB10] Melinda Tóth and István Bozó. Building dependency graph for slicing Erlang program. In *Conference of PhD Students in Computer Science: volume of extended abstracts CSCS 2010*, page 73, Szeged, 2010. Szegedi Tudományegyetem Informatikai Tanszékcsoport.

[TM15] Kozsik Tamás and Tóth Melinda. Let's PaRTE! *COMPUTERWORLD SZÁMÍTÁSTECHNIKA*, XLVI(2):16–17, 2015.

[TO15] Melinda Tóth and Gábor Oláh. Using the open-source RefactorErl in Ericsson, 2015. Talk at Functional Meetup, CRAFT 2015 Edition.

[TPRR15] Melinda Tóth, Attila Páter-Részeg, and Gordana Rakic. Introducing support for Erlang into SSQSA framework. In Theodore Simos and Charalambos Tsitouras, editors, *Proceedings of the International Conference on Numerical Analysis and Applied Mathematics 2014 (ICNAAM-2014): 4th Symposium on Computer Languages, Implementations and Tools, AIP Proceedings*, volume 1648. AIP Publishing, 2015. Art. No.: 310012, 4 p.

[VGT17] Ana Vrankovic, Tihana Galinac Grbac, and Melinda Tóth. Comparison of software structures in Java and Erlang programming languages. In Zoran Budimac, editor, *Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, CEUR Workshop Proceedings*, volume 1938. CEUR-WS.org, 2017. Art. No.: 18, 8p.

A. függelék

Adatfolyam elemzés

A függelékben szereplő cikkek [TB12b, TBHT10] végleges változatának publikációs adatai:

- Melinda Tóth and István Bozó. Static analysis of complex software systems implemented in Erlang. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 440–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

- Melinda Tóth, István Bozó, Zoltán Horváth, and Máté Tejfel. First order flow analysis for Erlang. In *8th Joint Conference on Mathematics and Computer Science, Selected papers, ISBN 978-963-9056-38-1*, pages 403–416, Komárno, Slovakia, July 2010.

Static Analysis of Complex Software Systems Implemented in Erlang *

Melinda Tóth, István Bozó

Eötvös Loránd University, Budapest, Hungary
{tothmelinda,bozoistvan}@caesar.elte.hu

Abstract. Static software analyser tools use different levels of intermediate source code representations that depend on the syntax and semantics of the language to be analysed. Most of the analyser tools use graph representation to efficiently retrieve information. Building such graphs for dynamically typed languages, such as Erlang, is not straightforward. In this paper we present static analysis methods to define the Dependency Graph representation of Erlang programs. The introduced methods cover the data-, control-, behaviour-flow and dependency analyses for sequential and parallel language constructs.

1 Introduction

Static analysis of the software products is a widely used technique to support different phases of the software development lifecycle. These analysis techniques can help in software development and maintenance tasks like: debugging, testing, code comprehension, cost estimation, model visualisation of programs, coding convention checking, or detecting possible errors. The common part of them is the analysis of the source code without actually executing the target program.

To perform a static analysis, an intermediate representation of the source code is required. The efficiency of the analysis highly depends on this representation. For this reason different intermediate source code representations have to be developed for different static analysis purposes. For instance, more detailed information is required for source code transformation and manipulation (in case of a refactoring tool) than for extracting the model of a live code. For source code transformation, beside the semantic information, the lexical and syntactic information is essential. For model extraction, only the high-level entities and the connection between them is required. Depending on the required information for the analysis, first we need to build the basic representation from the source code (e.g. AST). Then we extend the basic representation with the information of higher-level of abstraction (e.g. semantic information).

In this paper we define various forms of intermediate source code representations for the programs written in Erlang [7]. The language was designed to develop highly concurrent, distributed, fault tolerant systems with soft real-time

* Supported by TECH_08_A2-SZOMIN08

characteristics such as telecommunication systems. The language is dynamically typed, which makes the static analysis even harder.

We introduce a Semantic Program Graph to represent lexical, syntactic and different semantic information about the source code. We also give a formal description of the language and formalise the rules for the building of Data-Flow Graph and Control-Flow Graph. The data-flow is analysed from different aspects: zeroth and first order analysis, and the concurrent data-flow through message passing. Besides the data-flow analysis, the paper covers the control dependency relations, and some examples are given to the usage.

The information derived from the presented analyses can be used in several applications. Dependency Graphs are widely used in program slicing algorithms to perform change impact analysis. We have defined the impact analysis for Erlang programs [4] to select the subset of the program containing those expressions that are potentially affected by a change on the source code. Based on the result we can determine the test cases that are affected by the change on the source code and should be rechecked.

The Dependency Graph includes the control and data dependencies among expressions. Based on these dependencies we can perform further analysis to find the parallelisable components that can be run in parallel efficiently and without high synchronisation costs. Hence we have to calculate strongly connected components on the dependency graph, and analyse the resulted components.

The presented intermediate source code representations and the result of the analyses are integrated to the source code analyser and transformer tool, RefactorErl [2, 3]. We briefly introduce the semantic query language of RefactorErl, which is applicable to query the result of the presented analyses during the software development, maintenance or testing. The data-flow analysis is also used in some of the refactoring steps. To ensure safe transformation, the source code has to be analysed and allow the transformation if every precondition holds. The changes have to be propagated in the source code, so data-flow analysis is required to detect those expressions where further transformations are necessary.

The paper is structured as follows. In Section 1.1, we introduce the syntax of Erlang programs. In Section 2, we present the Semantic Program Graph to represent the Erlang programs. In Section 3, we describe the data-, behaviour- and control-flow graph building rules and further analysis based on the built graphs: data-flow reaching, concurrent data-flow analysis, control dependency analysis, program slicing. Section 4 describes the static analysis in RefactorErl and a query language to support querying the result of the presented analysis by the user. Section 5 discusses related work and Section 6 concludes the paper.

1.1 The Syntax of Erlang Programs

Our research focuses on the Erlang programs. Erlang is a dynamically typed functional programming language that was designed for building highly concurrent, fault-tolerant, distributed systems with soft-real time characteristic. In its syntax the functional style is mixed with some Prolog like elements.

In addition to the functional language constructs, the language has built in support for concurrency. The Erlang Virtual Machine handles the light-weight processes that communicate with asynchronous message passing.

We formalised the rules of our static analysis and the built graphs according to the syntax and semantics of language elements. We introduce the detailed syntax description of the language in Appendix A, and in Figure 1 we give a short overview of it.

An Erlang function (F) contains several function clauses. Each clause introduces the name of the function, the formal parameters of the function (patterns), optionally a guard expression and a sequence of expressions to be executed. The expression (E) types are detailed in Appendix A. In Figure 1 we show the syntax of the match expression ($P = E$), the tuple constructor ($\{E, ..., E\}$) and the function call ($E(E, ..., E)$). Patterns (P) are restricted to constant values, variables and tuple and list selectors.

V ::= variables (including the underscore pattern (_))
A ::= atoms
I ::= integers
K ::= $A \mid I \mid$ other constants (e.g. string, float, char)
P ::= $K \mid V \mid \{P,\ldots,P\} \mid$ `[`$P,\ldots,P$`|`P`]`
F ::= A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$`;`
$\vdots$
A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$`.`
E ::= $K \mid V \mid \{E,\ldots,E\} \mid E_{List} \mid P$ `=` $E \mid E$`(`$E,\ldots,E$`)` | ...

Fig. 1. Erlang syntax (partial)

2 Source Code Representation

Different source code analysis techniques exist, and the most common part of them is the usage of some intermediate source code representation for the analysis.

The most simple and the most current representation is the Abstract Syntax Tree (AST) of the program. The AST of a program contains the syntactic structure of the program without representing every detail about the source code. The main disadvantage of using an AST in further program analysis is the high cost of information retrieval: in most cases a whole AST traversal is needed to gather the required information about the source code. For instance, if we want to know where a function is called, you have to scan the AST of every module. Therefore, we choose a graph to represent the syntax and also the semantic information about the source code.

2.1 Semantic Program Graph

The Semantic Program Graph (SPG) is a rooted, directed, labelled and indexed graph that represents the Erlang source code in three different layers:

- *Lexical layer* – contains the token information about the source code. This layer stores information about the whitespaces and comments, and contains both the original and the preprocessed version of tokens.
- *Syntactic level* – contains the syntax tree of the source code.
- *Semantic layer* – contains extra calculated semantic information about the source code, such as module, function references, variable binding.

The SPG contains **nodes** and **directed edges** among the nodes. There are different node **classes** and edge types. The graph has a special starting node, the *root* node, the only element of the root node class. The root node is the starting point of the most of the queries on the graph. The other nodes of the SPG are the lexical (*lex*), syntactic (*file, form, clause, expr, typexp*) and semantic (*module, func, record, field, variable*) entities of the language. Every graph node has a set of **attributes** based on its class, e.g. the *function* has a *name* and an *arity* attribute. The directed edges represent the relations among the language entities. Each edge is **labelled** to represent different kinds of the certain relations between the nodes, thus we can say that the labels are the types of the edges. Each edge type points from a certain graph node class to another node class. For instance, the edge *moddef* links a *module* semantic node to a *file* syntactic node. The edges are also **indexed**, so links with the same tag and starting from the same node are maintained in their order.

We have defined the `my` module in Figure 2. The module contains a macro (`EOL/1`) and a function (`f/1`). The macro `EOL` has a string parameter and it simply appends a newline character to the end of the string. The function `f` has a parameter `S` and it calls the `put_chars` function from the `io` module with its parameter `S`.

```
-module(my).

-define(EOL(X), X ++ "\n").

f(S) ->
    io:put_chars(?EOL(S)).
```

Fig. 2. Example of Erlang source code

The syntax tree of the module `my` is shown in Figure 3. The syntax tree is built from the preprocessed source code, so all the macro applications are substituted (as {expr, 8} shows it in Figure 3).

The syntax tree is the base of the SPG. The syntactic and semantic levels of the SPG are shown in Figure 4. The oval boxes form the syntactic level of the

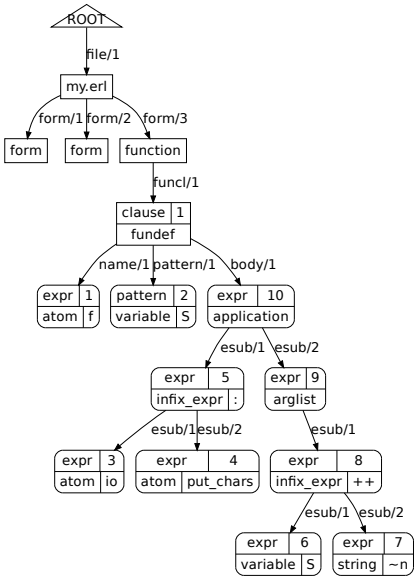


Fig. 3. Syntax tree for module `my`

graph (that contains the syntax tree of the program) and the hexagonal nodes and the dashed links form the semantic level of the SPG. There are semantic nodes for each defined and referred module (`my, io`), function (`f, put_chars`) and variable (`S`). These nodes are linked to the syntactic nodes to represent the definition or the reference to these entities (e.g. `varref, varbind, moddef`, `fundef`).

2.2 Building the Semantic Program Graph

The first step in creating the SPG is to build the syntax tree. The syntax tree is built of the scanned token list of the program. Before building the syntax tree we have to take into account the preprocessor directives and perform the file inclusion and macro substitution. We build the syntax tree of the programs from the preprocessed tokens. However, scanners and parsers can be generated based on the grammar of the language, but the preprocessing mechanisms in most of the cases are hard coded to the system.

The necessary information for the building of the semantic level of the graph is calculated by traversing the AST. The AST (and the SPG built from the AST) can be traversed by using **path expressions**. A path expression is a sequence of graph edge labels to be followed from a starting node. For instance, if we want to find the defined functions in the system, we have to start the query

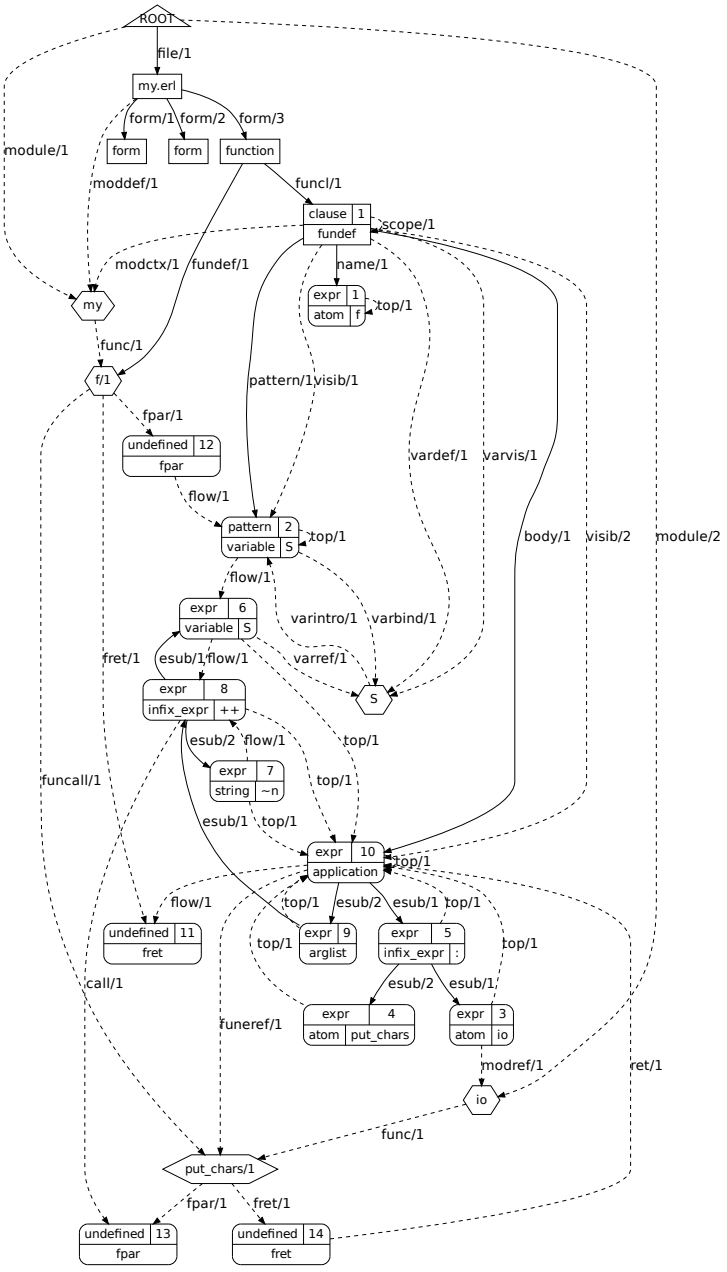


Fig. 4. Semantic Program Graph – Syntactic and Semantic Level

from the root node, and ask first the defined modules (*module* edge) and from the modules we can ask the defined functions of the modules (*func* edge), i.e. we have to follow the *[module, func]* sequence of labels from the root node. The syntax of path expressions is described in Figure 5. The main advantage of the graph representation is that the most frequently used queries have a fixed length, and there is no need to traverse the whole graph.

```
path() = [PathElem]

PathElem = Tag | {Tag, Index} | {Tag, Filter}
Tag      = atom() | {atom(), back}
Index    = integer() | {integer(), integer()} | {integer(), last}
Filter   = {Filter, 'and', Filter} | {Filter, 'or', Filter} |
           {'not', Filter} | {Attrib, Op, term()}
Attrib   = atom()
Op       = '==' | '/=' | '=<' | '>=' | '<' | '>'
```

Fig. 5. The syntax of *path expression*

The structure of the *path expressions* and the filters are written according to the Erlang EDoc type specification syntax [6] in Figure 5. The type `path()` is a sequence of `PathElem`. The `PathElem` can be a graph edge label (*Tag*) or a graph edge label with filtering options ({*Tag, Index*} or {*Tag, Filter*}). The former case represents the labelled graph edges to follow during graph traversal, and the latter one makes it possible to select a subset of graph nodes during the graph traversal according to the given filtering options. It is possible to filter the result with syntactic or semantic information (`{Tag, Filter}`) and also with the indices of edges in the graph (`{Tag, Index}`). For instance, the pair `{esub, {6,8}}` denotes the sixth, seventh and eighth subexpressions of a graph node. The graph edges can be traversed both forward (`Tag = atom()`) and backward direction (`Tag = {atom(), 'back'}`).

The building of the semantic level of the graph has two phases: to gather information about the source code and to add new semantic nodes and edges to the graph. Different semantic analysers can add several kinds of semantic information about the source code:

- *Module analyser* – adds a new semantic module node to the graph when a module definition or reference is found in the syntax tree and links them to the node (*moddef, modref*). Once a module node is added for a specific module, its definition or its references are linked to this node. Each module is linked to the *root* node (*module*).
- *Function analyser* – adds a new semantic function node to the graph when the first reference or the definition of the function is found in the syntax tree and links it to the semantic node. Afterwards, each found reference is linked to the semantic function node. For instance, local calls are linked with

the *funlref* labelled edge, external calls are linked with *funeref* edge, import or export references are linked with *funexp, funimp* edges. The semantic function node is also linked to the semantic module node of the defining module (*func*) and the defining syntactic function form (*fundef*).

- *Record analyser* – similar to the function analyser, creates a semantic record node and links the definition and references to it (*recref, recdef*). The analyser creates semantic record field nodes to each field given in the record definition and links them with their references and definitions (*fielddef, fieldref*). The semantic field nodes are linked to the defining semantic record node (*field*) and the semantic record node is linked to the semantic module node of the containing module (*record*).
- *Context analyser* – to help in further analysis and speed up queries on the graph, the SPG introduces some structural context edges. Every clause has a link to its direct subexpressions (*visib*), and each compound expression has a *clause* edge to its clauses. The subexpressions are linked to their topmost super-expression with *top* edges. The clauses also have a hierarchy as the subexpressions do. Each clause is linked to its containing scope clause with a *scope* edge, and the scope clauses are linked to the containing function clause with a *functx* edge.
- *Variable analyser* – variables are analysed upon their containing scope information. Once a variable is found in the syntax tree, its scope has to be determined. The semantic variable node is linked to its scope clause with a *vardef* edge and it is linked to every clause where it is visible (*varvis*). The references and the bindings are linked to the variable node with *varref, varbind* edges.

Further semantic analysis can be performed based on the syntax tree and the listed semantic information, such as interprocedural data-flow analysis, dynamic function call analysis or control-flow analysis.

3 Source Code Analysis

Based on the syntactic and semantic information stored in the Semantic Program Graph of Erlang programs, further analyses can be applied to calculate flow or dependency information. In this section we present data-, control- and behaviour-flow analysis towards creating a Dependency Graph of Erlang programs.

When we have defined the Data-Flow, Control-Flow and Dependency Graphs, we had to consider the main features of the language. Some properties of the language make the analysis simpler (such as the limited number of data constructors and selectors, or the single assignment variables), but there are more properties that make the analysis complicated (the lack of documented evaluation strategy, the dynamic nature of the language: dynamic function calls, dynamic process starting, communication via message passing).

3.1 Data-Flow Analysis

The *Data-flow analysis* is a technique for gathering information about how a program manipulates its data and what are the possible sets of values calculated at various points in a program. Several classical data-flow analysis applications exist, such as constant-propagation analysis, liveness analysis, available expression analysis, reaching definition analysis, etc. In case of the *Reaching Definition Analysis* we are interested in each program point, which assignments may have been made and not overwritten, when execution reaches this point along some path [14], so we should statically determine which definitions may reach a certain point in the program.

Erlang is a single assignment language, so we are interested in reaching definition analysis and finding those program points which value can be a copy of a certain expression or variable. Therefore, in this section we introduce the reaching definition data-flow analysis. The analysis builds up the *Data-Flow Graph (DFG)* of an Erlang program. The DFG contains direct data-flow information among expressions and the reaching relation defines the direct and indirect data-flow among them.

The DFG is a part of the Semantic Program Graph of RefactorErl. The analysis adds data-flow edges to the SPG based on the syntax and semantics of the language. The DFG is a directed labelled graph (DFG = (N, E)), its nodes are the Erlang expressions ($n_i \in N$) and its edges represent the direct data-flow among them.

We can distinguish four kinds of data-flow edges:

- $\xrightarrow{\mathbf{f}}$ **(flow edge)**: $n_1 \xrightarrow{\mathbf{f}} n_2$ represents that the result of n_2 can be a copy of the result of n_1. Their value is exactly the same, and changing the value of n_1 results in the same change of the value of n_2.
- $\xrightarrow{\mathbf{c_i}}$ **(constructor edge)**: $n_1 \xrightarrow{\mathbf{c_i}} n_2$, represents that the result of n_2 can be a compound value that contains n_1 as the ith element.
- $\xrightarrow{\mathbf{s_i}}$ **(selector edge)**: $n_1 \xrightarrow{\mathbf{s_i}} n_2$, represents that the result of n_2 can be the ith element of the compound data n_1.
- $\xrightarrow{\mathbf{d}}$ **(dependency edge)**: $n_1 \xrightarrow{\mathbf{d}} n_2$, represents that the result of n_2 can directly depend on the result of n_1.

We build the DFG with a compositional syntax based on the formal data-flow rules. These rules are presented and detailed in Appendix B in Figures 24 – 26. During the data-flow analysis we traverse the syntax tree part of the SPG and try to apply one of the data-flow rules. When a syntactic element matches to a left hand side of a rule we apply the right hand side of that rule and add the given edges to the graph. This rule-based graph building method results in the *Interprocedural Data-Flow Graph* containing the direct data-flow edges. The indirect data-flow can be obtained by traversing the DFG and calculating the transitive closure of the graph. We can define this closure with the *Data-Flow Reaching* relation. We have defined the zeroth order (Section 3.2) and the first order reaching (Section 3.5).

To give the basic idea behind the rules we show some example rules in Figure 6. The (*Variable*) rule describes that the value bound to a certain variable flows to all occurrences of the same variable. Rules (*Tuple exp.*) and (*Tuple pat.*) describe the constructor and selector operations of tuples. The edge c_i denotes that the value is the i^{th} element of the tuple, and s_i means that we select the i^{th} element of the compound pattern. The (*Fun. call*) rule defines the data-flow from the actual parameters to the patterns of the function definition, and the flow of the result of the function back to the function application.

	Expression	Direct Graph Edges
(Variable)	p binding of a variable n occurrence of a variable	$p \xrightarrow{\mathbf{f}} n$
(Tuple exp.)	e_0: $\{e_1, \ldots, e_n\}$	$e_1 \xrightarrow{\mathbf{c_1}} e_0, \ldots, e_n \xrightarrow{\mathbf{c_n}} e_0$
(Tuple pat.)	p_0: $\{p_1, \ldots, p_n\}$	$p_0 \xrightarrow{\mathbf{s_1}} p_1, \ldots, p_0 \xrightarrow{\mathbf{s_n}} p_n$
(Fun. call)	e_0: $\mathsf{m} : \mathsf{g}(e_1, \ldots, e_n)$ or $\mathsf{g}(e_1, \ldots, e_n)$ m:g/n: $\mathsf{g}(p_1^1, \ldots, p_n^1)$ when g_1 $\rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{g}(p_1^m, \ldots, p_n^m)$ when g_m $\rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_1 \xrightarrow{\mathbf{f}} p_1^1, \ldots, e_1 \xrightarrow{\mathbf{f}} p_1^m$ $\vdots$ $e_n \xrightarrow{\mathbf{f}} p_n^1, \ldots, e_n \xrightarrow{\mathbf{f}} p_n^m$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_m}^m \xrightarrow{\mathbf{f}} e_0$

Fig. 6. Static Data-Flow Graph building rules

3.2 Zeroth Order Data-Flow Reaching

We have already defined the direct edges of the Data-Flow Graph, but we have to define the indirect data-flow relation based on this graph as well. The *Data-Flow Reaching* is defined by the $\overset{\mathbf{0f}}{\rightsquigarrow}$ relation, where $n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2$ means that the value of the expression represented by n_1 in the DFG can be a copy of the value of n_2 – their values are equal. During defining the reaching we have to consider the followings:

- $n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_1$ always holds, because the value of an expression reaches itself (reflexive rule)
- If there is a flow edge $\xrightarrow{\mathbf{f}}$ between nodes n_1 and n_2, then the value of n_1 reaches n_2 (f rule)
- A compound data structure preserves the data in its elements. When we put an element n_1 into a data structure n_2 and the compound data reaches

another node n_3 and we take out the element from the compound data to n_4, then the packed value n_1 reaches n_4 (c-s rule)
– If the value of an expression n_1 reaches n_2 and the value of n_2 reaches n_3, then the value of n_1 reaches n_3, their values are equal (transitive rule)

Based on these, we can formalise the zeroth order data-flow reaching. We call it zeroth order, because it does not handle any context information, for instance about the calling context of functions.

Definition 1: Zeroth order data-flow relation The zeroth order data-flow reaching relation ($\overset{\mathbf{0f}}{\rightsquigarrow}$) is the minimal relation that satisfies the following rules:

$$n \overset{\mathbf{0f}}{\rightsquigarrow} n \qquad \text{(reflexive)}$$

$$\frac{n_1 \overset{\mathbf{f}}{\rightarrow} n_2}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2} \qquad \text{(f rule)}$$

$$\frac{n_1 \overset{\mathbf{c_i}}{\rightarrow} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3,\ n_3 \overset{\mathbf{s_i}}{\rightarrow} n_4}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_4} \qquad \text{(c-s rule)}$$

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_3} \qquad \text{(transitive)}$$

For some application of the data-flow reaching the relevant information is the last element of a flow chain. Thus, we introduce the forward and backward compact data-flow reaching. For instance, when applying the data-flow analysis for the dynamic function call detection [8], we have to detect whether the value of a variable is unambiguously defined in the source code or it can be the result of some operation.

Definition 2: Zeroth order compact forward data-flow relation The compact forward data-flow reaching ($\overset{\mathbf{0f_{cf}}}{\rightsquigarrow}$) is the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ \nexists n_3,\ n_3 \neq n_2 :\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{0f_{cf}}}{\rightsquigarrow} n_2} \qquad \text{(f-compact)}$$

Definition 3: Zeroth order compact forward data-flow relation The compact backward data-flow reaching ($\overset{\mathbf{0f_{cb}}}{\rightsquigarrow}$) is the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ \nexists n_0,\ n_0 \neq n_1 :\ n_0 \overset{\mathbf{0f}}{\rightsquigarrow} n_1}{n_1 \overset{\mathbf{0f_{cb}}}{\rightsquigarrow} n_2} \qquad \text{(b-compact)}$$

3.3 Zeroth Order DFG and Reaching Example

Let us consider the following Erlang module in Figure 7. The function `swap/2` swaps the values of a two-tuple. The function `get_1st/1` takes a tuple as an argument and swaps its values, then returns the first element of the swapped tuple. Finally, the function `cons/0` calls the function `get_1st/1` with the actual parameter `{1,2}` and returns the result of the function call.

```
-module(dataflow).

swap({A, B}) ->
    {B, A}.


get_1st(X) ->
    {E1, E2} = swap(X),
    E1.

const()->
    Y = get_1st({1,2}),
    Y.
```

Fig. 7. Example of Erlang code

The Data-Flow Graph of this module is shown in Figure 8. The result of the function `cons/0` is the value of the variable `Y`. It can be traced in the graph that the constant `2` can be the value of `Y`: $e_{25} \overset{\mathbf{0f}}{\rightsquigarrow} e_{32}$ (the notation e_i denotes the DFG graph node and i is the index of nodes in Figure 8). We pack the integer `2` into the tuple as its second element and pass the tuple to the argument of the function `get_1st/1` that also passes that value to the parameter of `swap/2`. The last function unpacks the values from the tuple and packs them into a new tuple in reverse order. Thus, the second element of the tuple (the integer `2`) becomes the first element. Then `get_1st/1` unpacks the resulted tuple and returns the first element of the tuple, that is the integer `2`. The function `cons/0` binds the result of the function call to the variable `Y` and returns that value. Thus, the result is the integer `2`.

Using the data-flow reaching relation we can formalise the traversal of the graph:

$$\frac{e_{22} \overset{\mathbf{f}}{\rightarrow} e_{32},\ e_{30} \overset{\mathbf{f}}{\rightarrow} e_{22},\ e_{20} \overset{\mathbf{f}}{\rightarrow} e_{30},\ p_{10} \overset{\mathbf{f}}{\rightarrow} e_{20}}{e_{22} \overset{\mathbf{0f}}{\rightsquigarrow} e_{32},\ e_{30} \overset{\mathbf{0f}}{\rightsquigarrow} e_{22},\ e_{20} \overset{\mathbf{0f}}{\rightsquigarrow} e_{30},\ p_{10} \overset{\mathbf{0f}}{\rightsquigarrow} e_{20}} \quad \text{(f rule (4 times))}$$

$$\frac{p_{10} \overset{\mathbf{0f}}{\rightsquigarrow} e_{20},\ e_{20} \overset{\mathbf{0f}}{\rightsquigarrow} e_{30},\ e_{30} \overset{\mathbf{0f}}{\rightsquigarrow} e_{22},\ e_{22} \overset{\mathbf{0f}}{\rightsquigarrow} e_{32}}{p_{10} \overset{\mathbf{0f}}{\rightsquigarrow} e_{32}} \quad \text{(transitive rule (3 times))}$$

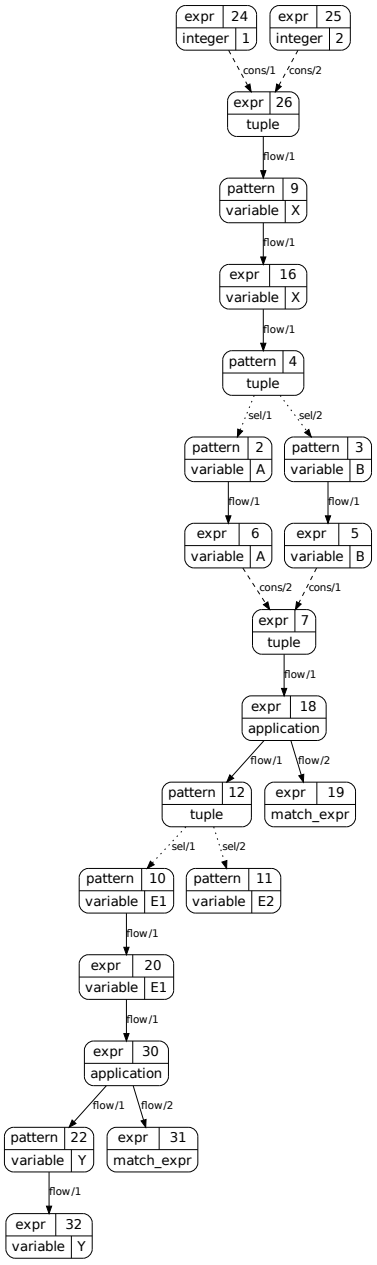


Fig. 8. Data-Flow Graph example

$$\frac{e_{18} \xrightarrow{\mathbf{f}} e_{12},\ e_7 \xrightarrow{\mathbf{f}} e_{18}}{e_{18} \overset{\mathbf{0f}}{\rightsquigarrow} e_{12},\ e_7 \overset{\mathbf{0f}}{\rightsquigarrow} e_{18}} \quad \text{(f rule (2 times))}$$

$$\frac{e_7 \overset{\mathbf{0f}}{\rightsquigarrow} e_{18},\ e_{18} \overset{\mathbf{0f}}{\rightsquigarrow} e_{12}}{e_7 \overset{\mathbf{0f}}{\rightsquigarrow} e_{12}} \quad \text{(transitive rule)}$$

$$\frac{p_2 \xrightarrow{\mathbf{f}} e_6}{p_2 \overset{\mathbf{0f}}{\rightsquigarrow} e_6} \quad \text{(f rule)}$$

$$\frac{e_{16} \xrightarrow{\mathbf{f}} p_4,\ e_9 \xrightarrow{\mathbf{f}} e_{16},\ e_{26} \xrightarrow{\mathbf{f}} e_9}{e_{16} \overset{\mathbf{0f}}{\rightsquigarrow} e_4,\ e_9 \overset{\mathbf{0f}}{\rightsquigarrow} e_{16},\ e_{26} \overset{\mathbf{0f}}{\rightsquigarrow} e_9} \quad \text{(f rule (3 times))}$$

$$\frac{e_{29} \xrightarrow{\mathbf{f}} e_9,\ e_9 \xrightarrow{\mathbf{f}} e_{16},\ e_{16} \xrightarrow{\mathbf{f}} p_4}{e_{26} \overset{\mathbf{0f}}{\rightsquigarrow} p_4} \quad \text{(transitive rule (2 times))}$$

$$\frac{e_6 \xrightarrow{\mathbf{c_1}} e_7,\ e_7 \overset{\mathbf{0f}}{\rightsquigarrow} e_{12},\ e_{12} \xrightarrow{\mathbf{s_1}} p_{10}}{e_6 \overset{\mathbf{0f}}{\rightsquigarrow} p_{10}} \quad \text{(c-s rule)}$$

$$\frac{e_{25} \xrightarrow{\mathbf{c_2}} e_{26},\ e_{26} \overset{\mathbf{0f}}{\rightsquigarrow} p_4,\ p_4 \xrightarrow{\mathbf{s_2}} p_2}{e_{25} \overset{\mathbf{0f}}{\rightsquigarrow} p_2} \quad \text{(c-s rule)}$$

$$\frac{e_{25} \overset{\mathbf{0f}}{\rightsquigarrow} p_2,\ p_2 \overset{\mathbf{0f}}{\rightsquigarrow} e_6,\ e_6 \overset{\mathbf{0f}}{\rightsquigarrow} p_{10},\ p_{10} \overset{\mathbf{0f}}{\rightsquigarrow} e_{32}}{e_{25} \overset{\mathbf{0f}}{\rightsquigarrow} e_{32}} \quad \text{(transitive rule (3 times))}$$

3.4 First Order Data-Flow Analysis

The zeroth order reaching is calculated based on the DFG. It does not consider the calling context of functions, thus the zeroth order reaching is an over-approximation (the DFG contains false positive data-flow edges). To address this problem we defined the *First order Data-Flow Reaching*. The first order analysis extends the DFG with context information about function calls to denote the entry point of the function from a given function call and the return point of the function to the same function call with the same index.

We will illustrate the calling context problem with an example and demonstrate how the first order analysis can avoid some false positive hints.

We can extend our previous example from Section 3.3 with another function, which calls the function `get_1st`. The result of the data-flow reaching changes according to the extension (Figure 9).

```
const2()->
    Z = get_1st({3,4}),
    Z.
```

In this case both the tuples `{3,4}` and `{1,2}` flow to the pattern `X`, and the result of the function `get_1st/1` can be either integer `2` or `4` after the swapping. The result of this function (e_{20}) flows to the applications (e_{30}, e_{41}), so these can be the values of variables `Y` and `Z`. However, it is obvious that when we call the function from `const/0`, the result is `2` and from `const2/0` the result is `4`.

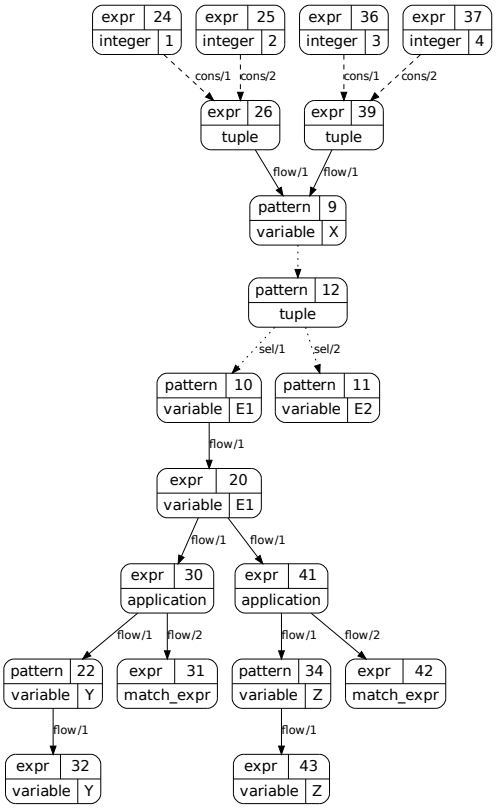


Fig. 9. Extended Data-Flow Graph example

The problem with the zeroth order data-flow graph is that it does not store any context information about the source code. Therefore, we can refine this analysis by adding some context information about the calling context of functions and distinguish the different function call and return points.

Extended First Order Rules To achieve the first order analysis, the 0^{th} order data-flow rules have to be extended with calling context information. Our

motivating example indicates that we should distinguish the different calls to a function. Therefore, we store additional information on the entry and exit points of a function rather than considering only the flow of the parameters.

Our first order analysis introduces new data-flow relations: $\overset{\mathbf{call(g)}}{\rightarrow}$ for entering and $\overset{\mathbf{ret(g)}}{\rightarrow}$ for leaving the function g. The context information is added to the edge as an index: $\overset{\mathbf{call(g,i)}}{\rightarrow}$ means the i^{th} call of the function, and $\overset{\mathbf{ret(g,i)}}{\rightarrow}$ means the return point of the i^{th} call. Figure 10 shows the compositional data-flow rule for a function call.

	Expressions	Graph edges
(Fun. call)	e_0: $\mathsf{m} : \mathsf{g}(e_1, \ldots, e_n)$ m:g/n: $\mathsf{g}(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{g}(p_1^m, \ldots, p_n^m)$ when $g_m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$. e_0 is the i^{th} analysed call to function $m : g/n$	$e_{l_1}^1 \overset{\mathbf{ret(g,i)}}{\rightarrow} e_0, \ldots, e_{l_m}^m \overset{\mathbf{ret(g,i)}}{\rightarrow} e_0$ $e_1 \overset{\mathbf{call(g,i)}}{\rightarrow} p_1^1, \ldots, e_1 \overset{\mathbf{call(g,i)}}{\rightarrow} p_1^m$ $\vdots$ $e_n \overset{\mathbf{call(g,i)}}{\rightarrow} p_n^1, \ldots, e_n \overset{\mathbf{call(g,i)}}{\rightarrow} p_n^m$

Fig. 10. First order function call rule

3.5 Extended First Order Data-Flow Relation

The presented data-flow rules describe direct flow information among Erlang expressions. Since data could flow from node n_1 to n_2, and from n_2 to n_3, we are curious whether the value of n_1 flows indirectly to n_3. The 1^{st} order data-flow relation returns those nodes in the graph where the value of a given node can flow through a sequence of the data-flow edges.

We derive the first order data-flow relation from the zeroth order data-flow relation considering the followings:

- $\overset{\mathbf{0f'}}{\rightsquigarrow}$ denotes the zeroth order data-flow relation calculated on the data-flow graph defined by the extended data-flow rules. The zeroth order flow relation operates on $\overset{\mathbf{f}}{\rightarrow}$ edges, but the first order function call rule replaces the corresponding $\overset{\mathbf{f}}{\rightarrow}$ edges with $\overset{\mathbf{call}}{\rightarrow}$ and $\overset{\mathbf{ret}}{\rightarrow}$ edges. The zeroth order reaching cannot use the new edges. This results in a smaller intrafunctional $\overset{\mathbf{0f'}}{\rightsquigarrow}$ relation.
- $\overset{\mathbf{1f}[\mu]}{\rightsquigarrow}$ denotes the first order data-flow relation.

- In the first order relation ($\overset{\mathbf{1f}[\mu]}{\rightsquigarrow}$) μ is a list of call ($\overset{\mathbf{call(g,i)}}{\rightarrow}$) and return ($\overset{\mathbf{ret(h,j)}}{\rightarrow}$) points. We have to record the names and the indices of the called functions, because later we have to find the corresponding exit points.
- Each node (n_i) that is reachable in the extended representation with the 0^{th} order data-flow relation is reachable by the first order relation (0^{th} *flow rule*).
- Similarly to the 0^{th} order relation, if a data constructor packs ($\overset{\mathbf{c_i}}{\rightarrow}$) the node n_1 into n_2 and the value of n_2 flows (with a first order flow) into the node n_3 and another data constructor unpacks ($\overset{\mathbf{s_i}}{\rightarrow}$) the value into n_4, then the value of n_1 flows into n_4 (1^{st} *c-s rule*).
- The call ($\overset{\mathbf{call(g,i)}}{\rightarrow}$) and the return ($\overset{\mathbf{ret(h,j)}}{\rightarrow}$) edges behave similarly as the flow $\overset{\mathbf{f}}{\rightarrow}$ edges, so the data flows through them (*call rule* and *return rule*).
- The data can flow through any function call (*call concat. rule*).
- If the value of the node n_1 flows into the node n_2 through the return value of a function call and the value of n_2 flows into the node n_3 through the return value of another function call, then the value of n_1 transitively flows into the node n_3 (*return concat. rule*).
- If we enter the function through the edge $\overset{\mathbf{call(g,i)}}{\rightarrow}$, then we have to leave the function through the $\overset{\mathbf{ret(g,i)}}{\rightarrow}$ edge (*reduce rule*) and leaving the function body through an $\overset{\mathbf{ret(g,j)}}{\rightarrow}$ ($j \neq i$) edge is not allowed (*Lemma* 3).

In Definition 4 we use the following notations:

- μ denotes a list;
- $hd(\mu)$ results the head (first) element of a list;
- $last(\mu)$ stands for the last element of a list;
- $\mu + +\rho$ denotes the concatenation of list μ and list ρ;
- μ_n denotes the n^{th} element of list μ.

Definition 4: First order data-flow relation The data-flow relation ($\overset{\mathbf{1f}}{\rightsquigarrow}$) is the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{\mathbf{0f'}}{\rightsquigarrow} n_2}{n_1 \overset{\mathbf{1f}[]}{\rightsquigarrow} n_2} \qquad (0^{th} \text{ flow rule})$$

$$\frac{n_1 \overset{\mathbf{c_i}}{\rightarrow} n_2,\ n_2 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_3,\ n_3 \overset{\mathbf{s_i}}{\rightarrow} n_4}{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_4} \qquad (1^{st} \text{ c-s rule})$$

$$\frac{n_1 \overset{\mathbf{call(g,i)}}{\rightarrow} n_2}{n_1 \overset{\mathbf{1f}[\mathbf{call}_{(\mathbf{g,i})}]}{\rightsquigarrow} n_2} \qquad (\text{call rule})$$

$$\frac{n_1 \overset{\mathbf{ret(h,j)}}{\rightarrow} n_2}{n_1 \overset{\mathbf{1f}[\mathbf{ret}_{(\mathbf{h,j})}]}{\rightsquigarrow} n_2} \qquad (\text{return rule})$$

$$\frac{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{1f}[\rho]}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{1f}[\mu{+}{+}\rho]}{\rightsquigarrow} n_3} \quad if\ (\exists f\ \exists i:\ (hd(\rho) = call_{(g,i)}))\ or\ \rho = []$$

(call concat. rule)

$$\frac{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{1f}[\mathbf{ret_{(h,j)}}|\rho]}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{1f}[\mu{+}{+}[\mathbf{ret_{(h,j)}}|\rho]]}{\rightsquigarrow} n_3} \quad if\ (\exists f\ \exists i:\ (last(\mu) = ret_{(g,i)}))\ or\ \mu = []$$

(return concat. rule)

$$\frac{n_1 \overset{\mathbf{1f}[\mu{+}{+}[\mathbf{call_{(h,i)}}]]}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{1f}[\mathbf{ret_{(h,i)}}]}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{1f}[\mu]}{\rightsquigarrow} n_3}$$

(reduce rule)

N^{th} Order Analysis. Based on the defined first order analysis, where we have stored the calling context in the DFG in one depth, we can generalise the second order analysis and store the calling context in two steps [19]. For example, in case of a higher order function (Figure 11), a dynamic function call in the body of the function depends on the parameter of the higher order call, thus it depends on the calling context of the higher order function. In Figure 11 we have defined the function $func/2$ that applies its first argument on its second argument: when we call this function from *call_pear*$/0$ it calls the function *pear*$/1$, and when we call this function from *call_apple*$/0$ it calls the function *apple*$/1$.

```
func(Fun, Data)->
    Fun(Data).
```

```
call_pear()->
    f(fun pear/1, [pear]).

call_apple()->
    f(fun apple/1, [apple]).
```

Fig. 11. Higher order functions

Based on this we can add the $\overset{\mathbf{call(pear,i)}}{\rightarrow}$ and the $\overset{\mathbf{call(apple,j)}}{\rightarrow}$ first order flow edges from the node of `Fun(Data)`, and also the $\overset{\mathbf{ret}*}{\rightarrow}$ edges. Calculating first order reaching on the DFG from the body of *call_pear*$/0$ results in that both function *apple*$/1$ and function *pear*$/1$ were called and their return values were reached. A solution to that problem could be to store two depth calling context: $\overset{\mathbf{call((func,1);(pear,i))}}{\rightarrow}$ and $\overset{\mathbf{call((func,2);(apple,j))}}{\rightarrow}$.

This analysis could be generalised iteratively to an n^{th} order flow analysis $(\overset{\mathbf{call((func_1,i_1);\ \ldots\ ;(func_n,i_n))}}{\rightarrow})$.

3.6 First Order DFG and Reaching Examples

The first order DFG of the previously mentioned example is shown in Figure 12. It can be traced that the integers `2` and `4` can be the result of the function

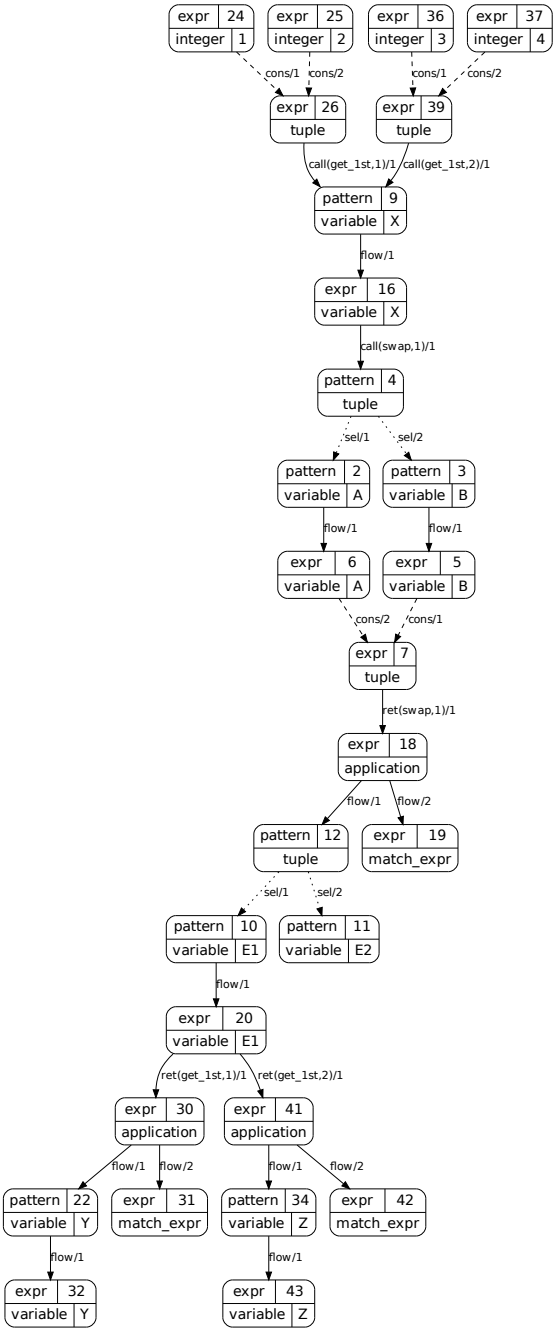


Fig. 12. First Order Data-Flow Graph example

`get_1st/1` ($e_{25} \overset{\mathbf{1f[call_{(get\_1st,1)}]}}{\rightsquigarrow} e_{20}$ and $e_{37} \overset{\mathbf{1f[call_{(get\_1st,2)}]}}{\rightsquigarrow} e_{20}$), but only the integer `2` can flow to variable `Y` (because of the *reduce rule*).

$$\frac{e_{25} \overset{\mathbf{1f[call_{(get\_1st,1)}]}}{\rightsquigarrow} e_{20},\ e_{20} \overset{\mathbf{1f[ret_{(get\_1st,1)}]}}{\rightsquigarrow} e_{32}}{e_{25} \overset{\mathbf{1f[]}}{\rightsquigarrow} e_{32}} \qquad \text{(reduce rule)}$$

3.7 Concurrent Data-Flow Analysis

Besides the function parameters there is another way to exchange data between functions (and also between different processes), that is message passing. Therefore, a naive approximation of calculating data-flow through message passing can be similar to the zeroth order data-flow through function calls. We can link each passed message to all of the receive expressions. That results a huge DFG containing lots of false flow edges. To avoid this, we should restrict the set of possible receivers of messages using context information about the message passing.

Concurrent Erlang Processes and Message Passing The language was designed for developing concurrent and distributed applications. Spawning a processes on remote nodes is as easy as spawning it on a single node. Since Erlang uses light-weight processes and processes are spawned at the virtual machine level, the spawning and destroying of processes is quite fast. The processes are separated from operating system processes and behave in the same way on every platform. The virtual machine takes care of spawning, destroying and scheduling of processes. The processes are independent and do not share memory, as they communicate only through message passing. Message passing is asynchronous and the messages arrive at a message queue of the process. Messages are waiting in a message queue until they are processed.

The following functions work in the same way on a local node, or even if there is a set of interconnected nodes. In this paper we will describe the message passing in case of a local node. The analysis can be extended easily to apply for distributed applications as well.

The basic language elements for concurrency are spawn, register, receive and send.

Spawning processes With the spawn function we can create new processes. We can spawn a lambda expression, an exported function on a local node, or even on the remote Erlang node. There are some versions for this BIF (Built in Function):

`spawn` – The function `spawn/n` spawns a separate process with the given function and returns its process identifier (PID). The creating process will not be notified if the created process has terminated abnormally. The function is available with different arities: `n=1,2,3,4`.

`spawn_link` – The function `spawn_link/n` spawns the given function, but creates a link between the parent and the spawned process. The creating process will be notified if the created process terminates abnormally and will cause to

crash the creating process if the exit is not trapped. The functions are available with different arities: `n=1,2,3,4`.

`spawn_monitor` – The function `spawn_monitor/n` spawns the given function and returns a tuple of a PID and a reference for the process. With this reference we can monitor the created process. The creating process will be notified if the created process terminates normally even if it crashes, but will not cause the creating process to terminate. The function is available with different arities: `n=1,3`.

Registering processes With the function `register/2` we can associate a local process identifier with a given name. The function will fail if there is a process registered with the same name, or the local process does not exist.

If we have several nodes connected and communicating with each other, we can also register the process globally with the function `global:register_name/n` (n=2,3).

If the process is registered it can be accessed by its name instead of the PID when sending it a message.

Sending messages Since the spawned processes are independent, the only way of communication is message passing, either through the network, or only on a local node. The process can be accessed by its process identifier obtained by spawning the process, or by the given name, if it has been registered. There are some alternatives for sending messages, but we describe only the most common cases, as the other functions behave similarly.

The right-associative infix send operator `!` (exclamation mark) and its function version `send/2` are used most commonly. The operator takes two operands: the right operand, which is the message to be send, and the left operand, which is the destination process. As a result, the send operator returns the message. The destination can be a PID, a registered process name, or even a registered process on a remote node.

The operator `!` and the `send/n` (n=1,2) send the message to the mailbox (or message queue) of the processes. To send a message to a globally registered process we can use the function `global:send/2`.

Receiving messages The messages arrive at the mailbox of the process. We can extract messages with the `receive` expression. The receive expression looks like a case expression (or switch in other languages), except it suspends until it can extract a message form the mailbox, or the given timeout has elapsed. We can extract messages with pattern-matching and proceed with execution according to the received message.

Detecting Spawned Processes Since the process identifiers are created dynamically or passed to the functions as parameters, statically detecting the recipient of a message is not straightforward and sometimes it is impossible to calculate. Let us consider the following Erlang function:

```
send_data(Pid) ->
    Data = do_some_computation(),
    Pid  ! {"Sending computed data", Data}.
```

In this example, if we do not have further knowledge about the function parameter `Pid`, we cannot detect where the actual message can flow. Thus we concentrate on the analysing of process identifiers and on calculating the set of functions that can be executed as processes. To solve the latter mentioned problem we have to analyse process spawning. For the sake of simplicity we describe in more detail only the function `spawn/3` hereinafter (the functions `spawn*/n` can be handled similarly):

```
Pid = spawn(ModName, FunName, ArgList)
```

The function `spawn/3` executes the given function in the new independent process and returns its unique identifier. The `spawn/3` function takes three arguments: the name of the implementing module, the name of the function to be spawned and the list of the actual parameters for this function. Thus the following triple `{ModName, FunName, length(ArgList)}` identifies the function that is spawned in a new process.

To calculate the possible values of the identifying triple we can use the sequential first order data-flow reaching defined in Section 3.4. The following sets define the possible values of the elements of the triple, where $mn \in V$ denotes the node representing the module name, $fn \in V$ denotes the node representing the function name and $arg \in V$ denotes the node representing the argument list in the Data-Flow Graph – $DFG = (V, E)$:

- $MN = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} mn, \nexists n', n' \in V, n' \neq n : n' \overset{\mathbf{1f}}{\rightsquigarrow} n\}$
- $FN = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} fn, \nexists n', n' \in V, n' \neq n : n' \overset{\mathbf{1f}}{\rightsquigarrow} n\}$
- $Arg = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} arg, \nexists n', n' \in V, n' \neq n : n' \overset{\mathbf{1f}}{\rightsquigarrow} n\}$

For instance, the set MN contains those expressions which values can flow into mn, i.e. we perform static backward data-flow reaching starting from mn. When the node mn is a variable, we need only the binding of the variable and neglect its references, because only the origin of the variable holds useful information for us. We select only the source nodes from the DFG, which has zero indegree.

Ideally, the type of the elements in MN and FN are atom and the type of the elements of Arg is an n-element list expression. In this case we can unambiguously identify the spawned function. Otherwise we can use some heuristic to narrow the possible set of functions that can be executed in the spawned process which receive the sent messages (for details see Section Heuristics for process detection).

Hereinafter SF_{Pid} denotes the set of functions that could be executed by the process `Pid`: $SF_{Pid} = \{\{val(M), val(F), size(A)\} | M \in MN, F \in FN, A \in Arg\}$

Function `val/1` returns the value of an expression instead of its node identifier in the DFG. Function `size/1` returns the estimated size of a list expression.

Lists have a variable length, so statically calculating the length of a list is not straightforward.

For the sake of simplicity we present the introduced sets in a simple example:

```
-module(mymod).

start(Fun, Args) ->
   Pid = spawn(?MODULE, Fun, Args).
   Pid ! start,
   Pid.

init() ->
   start(loop1, [init, []]).

process(Data) ->
   start(loop2, [proc, Data]).

loop1(State, Data) ->
   ...
loop2(Tag, Data) ->
   ...
```

The presented Erlang module `mymod` defines the function `start/2`, which spawns a process and then sends a message to the newly spawned process. The name and the parameters of the function to be executed by the process are the parameters of the function `start/2`, so they are not defined in the function. The first parameter of `spawn/3` is given by a predefined macro application `?MODULE`, the substitution of that macro is the name of the current module `mymod`. The function `init/0` calls the function `start/2` with the actual parameters `loop1` and a two-element list. The function `process/1` calls `start/2` with the actual parameters `loop2` and a two-element list. The body of the function `loop*/2` is not important yet.

Using backward first order data-flow analysis we obtain the following node sets, where $\$Expr\$$ means the node representing expression $Expr$ in the data-flow ($\$Expr\$ \in V$):

- $MN = \{\$?MODULE\$\}$
- $FN = \{\$loop1\$, \$loop2\$\}$
- $Arg = \{\$[init, []]\$, \$[proc, Data]\$\}$
- $SF_{Pid} = \{\{mymod, loop1, 2\}, \{mymod, loop2, 2\}\}$

We have identified two possible functions (SF_{Pid}) that can be spawned in `start/2`, and these are the recipients of the message passed in expression `Pid ! start`.

Since we can refer to any registered processes with the associated alias, we have to analyse the calls of function `register/2` too:

```
_True = register(Alias, PidExpr)
```

In this case we have to calculate the possible values of the expression `Alias` and to identify the function that has been spawned in a process with identifier `PidExpr`. To detect the possible values of these expressions, in both cases we require backward data-flow reaching:

- $AN = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} an, \nexists n', n' \in V, n' \neq n : n' \overset{\mathbf{1f}}{\rightsquigarrow} n\}$
- $Atom_{AN} = \{n \in V \mid \exists n',\ n' \in AN,\ \exists n'',\ n'' \in V,\ n'' \overset{\mathbf{1f}}{\rightsquigarrow} n,\ val(n'') = val(n),\ type(n') = atom,\ n \in MPass\}$
- $PN = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} pn, type(n) = spawn\_app\}$

In the former sets an denotes the node representing the `Alias` in the data-flow graph; pn represents the process identifier expression `PidExpr` in the data-flow graph and $MPass$ represents the elements of the message passing expression. The function `type/1` returns the type on an expression; $type(n) = spawn\_app$ means that node n is an application of function `spawn`.

To calculate the function executed in the process referred by the given `Alias` we have to select the nodes representing a function call to `spawn*/n`. This set of nodes is denoted by PN. We have to calculate the possible functions to be executed by the processes SF_{Pid_i} for every element of PN. Since the name `Alias` could refer to all of these processes, SF_{Alias} will denote the union of these sets.

The main point in registering a process with an atom *atom* alias is that the registered process can be accessed with the registered name *atom* at different points of the program without having any information about its PID. Thus an atom used in a message passing may refer to a process spawned in another function from another module, even if there is no data-flow connection between them. Therefore, we have to identify the atoms in message passing that could refer to the same process: $Atom_{AN}$. The elements of $Atom_{AN}$ should identify the same triples as the `Alias`: $\forall A \in Atom_{AN} : SF_A = SF_{Alias}$.

Consider the modified version of the previous example, where the function `init/1` registers the spawned process with a given alias `Alias`. The function `reg_proc` calls `init/1` with the actual parameter `proc1` and then sends a message using the registered alias.

```
start(Fun, Args) ->
   Pid = spawn(?MODULE, Fun, Args).
   Pid ! start,
   Pid.

init(Alias) ->
   P = start(loop1, [init, []]),
   register(Alias, P).

loop1(State, Data) ->
   ...
```

```
reg_proc() ->
   init(proc1),
   proc1 ! some_message.
```

We can calculate that the atom `proc1` refers to the function `mymod:loop1/2` in message passing expressions:

- $AN = \{\$proc1\$\}$
  - from function call `init(proc1)`
- $Atom_{AN} = \{\$proc1\$\}$
  - from expression `proc1 ! some_msg`
- $PN = \{\$spawn(?MODULE, Fun, Args)\$\}$
- $SF_{Alias} = SF_{Pid} = \{\{mymod, loop1, 2\}\}$
- $SF_{proc1} = SF_{Alias} = \{\{mymod, loop1, 2\}\}$

Heuristics for process detection Ideally the sets MN, FN and Arg contain atom nodes and list nodes with finite length, but it is not the case for industrial sized code. Therefore, we have studied some heuristics that can help to detect the possible functions to be executed in a process.

These heuristics are approximations of concurrent data-flow. The value of a variable representing the module name, the function name or the parameter list often cannot be calculated statically, therefore we want to approximate them based on the analysed source code. These heuristics are approximations of the dynamic/runtime information that is not available at compile time.

For instance, we can calculate the name of the module and the function to be spawned, but we cannot calculate the length of the parameter list. Since we have analysed the modules and functions before the message passing analysis, we can search for functions in the module without regarding the arity of the function. If we have found only one function, this must be the spawned function, otherwise there are more candidates to be spawned and we will scan each function body for the corresponding receive expressions.

These kinds of heuristics over-approximate the concurrent data-flow edges and the resulted DFG contains some false positive hints.

The used heuristics are based on the partial knowledge about the module name, function name and arity, and we extend this knowledge with information about the source code base. The used heuristics are:

1. when the name of the module (m) and the name of the function (f) are atoms – we select all functions with the name f from the module without regarding its arity n_i and we add $\{m, f, n_i\}$ to SF_*;
2. when the name of the module (m) is an atom – we select all functions from the module without regarding its name f_i and its arity n_i and we add $\{m, f_i, n_i\}$ to SF_* for each function f_i/n_i;

3. when the name of the module (m) is an atom and we can calculate the length of the parameter list (a) – we select all functions f_i from the module with the calculated arity a and we add $\{m, f_i, a\}$ to SF_*;
4. when the name of the function (f) is an atom and we can calculate the length of the parameter list (a) – we select every module (m_i) that defines a function `f/n` and we add $\{m_i, f, a\}$ to SF_*.

It is possible to use other heuristics (for instance, when only the arity of the function is known), but most of them result in a huge set of possible functions and thus we should generate lots of edges to the Data-Flow Graph.

Consider the following variation of our example, when the name of the function is a parameter of the function `init/2`.

```
start(Fun, Args) ->
   Pid = spawn(?MODULE, Fun, Args).
   Pid ! start,
   Pid.

init(Alias, FunName) ->
   P = start(FunName, [init, []]),
   register(Alias, P).

loop1(State, Data) ->
   ...
loop2(Tag, Data) ->
   ...
```

We can deduce that:

- $MN = \{\$?MODULE\$\}$
- $FN = \{\$FunName\$\}$
- $Arg = \{\$[init, []]\$, \$[proc, Data]\$\}$

Since the name of the function is unknown, we should use a heuristic. The name of the module is `mymod` and the arity of the function is `2`, so we can use the first heuristic from the listing. We are searching for the described functions (f_i) in the module: `loop1/2` and `loop1/2`, and we add $\{\{mymod, loop1, 2\}, \{mymod, loop2, 2\}\}$ to SF_{Pid}.

Data-Flow through Message Passing In the followings we concentrate on message passing expressions using the send operator (`!`): e_1 ! e_2. The left subexpression is a process identifier or an alias of a registered process stored in a variable or a simple atom. The right subexpression is the message to be sent.

The built-in function `send/2` can be analysed similarly. For the sake of simplicity we do not explain the case when the origin of the recipient of the message passing is a registered process on another node, because it is possible to extend our analysis to handle it.

	Expressions	Graph edges
(Send exp.)	e_0: e_1 ! e_2 e': receive p_1 when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \rightarrow$ $e_1^n, \ldots, e_{l_n}^n$ after $e \rightarrow e_1, \ldots, e_s$ end	$e_2 \overset{\mathbf{f}}{\rightarrow} e_0$ $\mathbf{e_2} \overset{\mathbf{f}}{\rightarrow} \mathbf{p_1}, \ldots, \mathbf{e_2} \overset{\mathbf{f}}{\rightarrow} \mathbf{p_n}$ $e_{l_1}^1 \overset{\mathbf{f}}{\rightarrow} e', \ldots, e_{l_n}^n \overset{\mathbf{f}}{\rightarrow} e'$ $e_s \overset{\mathbf{f}}{\rightarrow} e'$

Fig. 13. Concurrent data-flow rule

To analyse a send expression we have to identify the recipient of the message, so we have to find the origin of the left-hand side subexpression e_1. We use backward data-flow reaching to find the spawn expression that returns the process identifier e_1 or to find the expression that is registered as e_1 (alias):

- $Spawn = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} e_1, type(n) = spawn\_app\}$
- $Reg = \{n \in V | n \overset{\mathbf{1f}}{\rightsquigarrow} e_1, type(sup(n)) = reg\_app\}$

`sup(n)` denotes the superior expression of node n, i.e. n is a subexpression of `sup(n)`. If the spawning ($Spawn$) or registering (Reg) expressions are found, we can use the previously defined algorithm to calculate SF_{e_1}.

When e_1 is an atom or the backward reaching from e_1 returns an atom, we cannot use reaching to detect the SF_{e_1}, hence there is no data-flow connection between the used alias and the registering expression. At this point of the analysis we need the sets AN and SF_A for every register expression where $A \in Atom_{AN}$, thus it is required to calculate them in a previous stage of the analysis. In this case we have to calculate the possible atom values of e_1 – $name_1, ..., name_k$ – and select SF_{name_i} from the previously constructed sets. In this case $SF_{e_1} = \bigcup_{i=1}^{k} SF_{name_i}$.

Message passing indicates data-flow edges between the sender and the receiver expression. Therefore, after identifying the possible functions (SF_{e_1}) we have to find the possible receiver expressions. We have to collect the receiver expressions from the body of the executed function and from the body of the functions that are transitively called from the executed function:
$Rec = \{n \in V \mid type(n) = receive\_expr,\ F \in tr\_closure(SF_{e_1}),\ n \in body(F)\}$

The function `tr_closure/1` returns the transitive closure of the $\overset{\mathbf{call}}{\rightarrow}$ relation, where $f_1 \overset{\mathbf{call}}{\rightarrow} f_2$ means that function f_1 calls function f_2 and f_i is represented by the previously defined triple. The function `body/1` returns the expressions from the body of a function.

We apply the data-flow rule presented in Figure 13 for every $e' \in Rec_{e_1}$ receive expression. The sent message e_2 flows into the different patterns $(p_1, ..., p_n)$ of the selected receive expression, i.e. $\mathbf{e_2} \xrightarrow{\mathbf{f}} \mathbf{p_i}$. The result of the receive expression can be the value of the last expression of its clauses ($e^i_j \xrightarrow{\mathbf{f}} e'$) and the result of the send expression is the message itself ($e_2 \xrightarrow{\mathbf{f}} e_0$).

Let us consider another extension of the previous example, where we extend the body of function `loop1/2` with a receive statement.

```
loop1(State, Data) ->
   receive
     start ->
       Data = initial_steps(),
       loop(started, Data);
     Msg ->
       NewData = process_msg(Msg, Data),
       loop(State, NewData);
     stop ->
       closing_steps()
   end.
```

There were two send expressions in the previous example: `Pid ! start` and `proc1 ! some_message`. In the former case we have to calculate the *Spawn* set: $Spawn = \{\$spawn(?MODULE, Fun, Args)\$\}$ and then $SF_{Pid} = \{\{mymod, loop1, 2\}\}$. There is only one receive expression in the body of the function `mymod:loop1/2`, so we link the sent message to its patterns: $\$start\$ \xrightarrow{\mathbf{f}} p$, where $p \in \{\$start\$, \$Msg\$, \$stop\$\}$. To analyse the latter send expression we have to calculate $SF_{proc1} = \{\{mymod, loop1, 2\}\}$, then find the receive expressions and create the link among the message and the patterns in the Data-Flow Graph: $\$some\_message\$ \xrightarrow{\mathbf{f}} p_i$.

Refining the Processes Analysis We overestimate the concurrent data-flow edges, since the introduced static concurrent data-flow calculation algorithm does not consider the order of the sent messages or the liveness of processes. Further analyses should be performed to refine the resulted graph. The order of sent messages should be stored as context information.

It is possible to unregister the name of a process in Erlang programs, and after unregistering the process name we cannot send a message to the process by referring to its name. To detect the liveness of processes at a given point of the program we should improve our concurrent data-flow analysis and we have to use the control-flow analysis (Section 3.9) to calculate the execution paths of the program. Similarly, *exit* signals also have to be considered.

Improving the 1^{st} Order Data-Flow Analysis Beforehand we have introduced the process and message passing analysis for Erlang. Both sections

assume that we have a data-flow reaching relation. Therefore, we split the data-flow graph building algorithm into two parts. In the first stage we calculate the sequential Data-Flow Graph based on the rules from Sections 3.1–3.4.

In the second stage we calculate the concurrent data-flow edges based on the analysis described in Sections *Detecting Spawned processes* and *Data-Flow through Message Passing*. This analysis extends the DFG with new data-flow edges, thus calculating 1^{st} order reaching could result in a more accurate result set. Therefore, running the process analysis on the refined concurrent data-flow graph could generate new data-flow edges. It is possible to run the process analysis algorithm iteratively until it reaches its fixed point. The algorithm terminates when there is no more new message passing expression that we can analyse or we found the recipient for every message passing expression. When the analysis terminates, the resulted graph contains the possible statically calculable data-flow connections among Erlang statements. Unfortunately, with static analysis and the used heuristics we cannot avoid false hits.

Since we do not introduce new edge types for the Data-Flow Graph (only $\xrightarrow{\mathbf{f}}$ edges are generated), the definition of the reaching relation remains the same as it was in the sequential case.

The following example demonstrates the necessity of the iterative application of the algorithm.

```
start() ->
   Pid1 = spawn(?MODULE, fun1, []),
   Pid2 = spawn(?MODULE, fun2, []).
   Pid1 ! {pid, Pid2}.

fun1() ->
   receive
       {pid, Pid} -> Pid ! some_message
   end.

fun2(Tag, Data) ->
   receive
       A -> do_sth(A)
   end.
```

The function `start/0` spawns two processes and sends the process identifier of the second process to the first process. The first process executes the function `fun1/0`, i.e. waits for a message that contains a process identifier and sends a message to the received `Pid`. The second process executes the function `fun2/0`, i.e. waits for a message and executes a function call `do_sth/1` after the message is received.

It is obvious that a backward reaching on the sequential data-flow graph does not find the origin of `Pid` in the message passing, so we cannot deduce that it refers to function `fun2/0`. However, we can perform the second stage of the data-flow analysis for the send expression `Pid1 ! {pid, Pid2}` and add a flow edge

between the sent message and the receive pattern in `fun1/0`: $\${pid, Pid1}\$ \xrightarrow{\mathbf{f}} \${pid, Pid}\$. Then by performing a backward reaching on the concurrent Data-Flow Graph we get the origin of `Pid` and we can deduce that it refers to `fun2/0`. Now, we can add the flow edge: $\$some_message\$ \xrightarrow{\mathbf{f}} \$A\$.

3.8 Behaviour-Flow Analysis

The behaviour-flow or Behaviour-Dependency Graph [18] describes a potential data related dependency among expressions. A data dependency relation between two graph nodes ($n_1 \rightsquigarrow n_2$) means that the behaviour of n_2 depends on the result/behaviour of n_1, so the change of node n_1 may have an impact on n_2. This kind of information is essential when we want to follow the evolution of software systems and help the developers to maintain the program. The result of this analysis can provide some information about the expressions (or functions/modules) that could be affected by a change on the source code. Based on this information the developer can decide whether the planned change on the source code is performable. The behaviour dependency relation can be computed using the data flow, data dependency and the behaviour dependency edges (described in [18]).

Definition 5. The behaviour dependency relation $\overset{\mathbf{b}}{\rightsquigarrow}$ is defined as the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{\mathbf{1f}}{\rightsquigarrow} n_2}{n_1 \overset{\mathbf{b}}{\rightsquigarrow} n_2} \qquad \text{(d-rule)}$$

$$\frac{n_1 \overset{\mathbf{b}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{b}}{\rightarrow} n_3,\ n_3 \overset{\mathbf{b}}{\rightsquigarrow} n_4}{n_1 \overset{\mathbf{b}}{\rightsquigarrow} n_4} \qquad \text{(b-rule)}$$

Definition 6. The data and behaviour dependency relation $\rightsquigarrow$ is defined as the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{\mathbf{1f}}{\rightsquigarrow} n_2}{n_1 \rightsquigarrow n_2} \qquad \text{(data-rule)}$$

$$\frac{n_1 \overset{\mathbf{1f}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{d}}{\rightarrow} n_3,\ n_3 \overset{\mathbf{b}}{\rightsquigarrow} n_4}{n_1 \rightsquigarrow n_4} \qquad \text{(b-dep-rule)}$$

To informally explain these definitions we use some simple expressions examples:

```
func(...) ->
    ...
    A = 1 + 2,
    X = A,
    B = A * A,
    ...
```

Definition 5 describes that data-flow and the behaviour dependency edges ($\overset{\mathbf{b}}{\rightarrow}$) propagate behaviour dependency among expressions (d-rule, b-rule). In our example the value of A reaches X, so when we change A that has an impact on X, changing the behaviour of A affects X too.

Definition 6 presents that data-flow reaching holds a special dependency, because those nodes from the Data-Flow Graph which could be a copy of node n_1 are affected by changing the value of n_1, so modifying n_1 could have an impact on them ($n_1 \rightsquigarrow n_2$).

Considering the expression `A = 1+2` we can notice that changing the expression `1` to an atom `something_else` results that the expression `something_else+2` could not be evaluated and that it results a run-time error. Then each expression which behaviour depends on the value of `1+2` could not be evaluated. Therefore, when there is a data dependency connection between two nodes ($n_1 \overset{\mathbf{d}}{\rightarrow} n_2$ – $1 \overset{\mathbf{d}}{\rightarrow} 1 + 2$), changing the data in n_1 may have an impact on the behaviour of n_2, and those nodes which behaviour may depend on n_2 (`B = A * A`), also may alter the behaviour from the same data change (b-dep-rule).

3.9 Control-Flow Analysis

The Control-Flow Graph (CFG) represents all the possible execution/evaluation paths of the program that can be chosen for every possible input. The CFG is a language dependent representation of the program as it is based on the semantics of the language.

We have defined control-flow rules for Erlang programming language based on its semantics. The language has strict evaluation, which means that before evaluating a compound expression, its subexpressions have to be evaluated. In every case the subexpressions are evaluated in left to right order. The defined control-flow rules are compositional, thus the graph can be composed from the previously computed subgraphs. We use the SPG of RefactorErl and use the same identifiers for the vertices in the CFG and we extend the set of nodes with some dummy vertices for joining branches, error nodes, etc. The rules are defined and described in more detail in Appendix C in Figures 29–32.

The notations in the figures are the followings: $e, e_i \in E$ are expressions, $g, g_j \in E$ are guard expressions, $p, p_k \in P$ are patterns and $f/n \in F$ stands for functions. The $e'_0 \in E$ is a dummy node in the control flow graph, which represents an entry point of the compound expression or a joining of dummy nodes (ret) to represent the return of conditional branching expressions. The relation $\rightarrow$ represents the control-flow between the nodes. The edges that have no labels represent sequences, and edges with labels represent:

- conditional branching and pattern matching with ($\overset{\mathbf{yes}}{\rightarrow}$), ($\overset{\mathbf{no}}{\rightarrow}$) edges
- returning to a previous expression ($\overset{\mathbf{ret}}{\rightarrow}$),
- function calls/applications with ($\overset{\mathbf{call}}{\rightarrow}$),
- receive expression with ($\overset{\mathbf{rec}}{\rightarrow}$),
- send expression with ($\overset{\mathbf{send}}{\rightarrow}$).

The relations ($\overset{\mathbf{call}}{\rightarrow}$), ($\overset{\mathbf{rec}}{\rightarrow}$), ($\overset{\mathbf{send}}{\rightarrow}$) represent special relations which indicate the possible dependency between the nodes of different functions (for details, see Section 3.10). In the rest of this section we describe a small example to give a general overview about the control flow in Erlang. The reader can find the listing and discussion of formal rules in Appendix C.

A simple CFG example The simple factorial computing function is described in Figure 14. The function gets a non negative number and returns its factorial. By definition the factorial of `0` is `1` and for larger number we can calculate the factorial by multiplying `N` with the factorial of `N-1`.

```
fact(0) -> 1;
fact(N) when N > 0 ->
    N * fact(N - 1).
```

Fig. 14. Definition of the factorial function

Figure 15 shows the CFG for this simple factorial function. The graph is built on the formal rules described in Appendix C.

The entry point of the function is the node `FORM(1)`. The actual parameter is matched against the first formal parameter `0`. If it succeeds, the $\overset{\mathbf{YES}}{\rightarrow}$ edge is followed and the constant `1` value is returned, otherwise the control flows to the next pattern through the $\overset{\mathbf{NO}}{\rightarrow}$ edge. As the next pattern is a variable, the pattern matching will succeed ($N \overset{\mathbf{YES}}{\rightarrow} N > 0$). The next step is to evaluate the guard expression ($N > 0$). If the guard expression holds for the actual value, the body of the function is evaluated. The programming language has strict evaluation, and the subexpressions are evaluated first in left-to-right order. First the left operand of the multiplication (N) is evaluated. As it is a variable, the evaluation may proceed to the next subexpression. After that, the right operand of the multiplication is evaluated, a function application $fact(N-1)$. As the function name may come form a compound expression that is evaluated in runtime, first the name of the function should be computed, and then the argument of the function application (analogously to the multiplication). The step between the return of the function application and the multiplication is marked with a special edge ($fact(N-1) \overset{\mathbf{funcall}}{\rightarrow} N*fact(n-1)$) as the evaluation of the function call may affect the return of the function (if the called function fails). This information will be used during the composition of the separately computed parts of the graph.

The graph includes a special error node `ERROR(form, 1)`, because the function is a partial function and may produce a runtime error if none of the patterns and guards match. In the current example the function fails, if it gets a negative number as an argument.

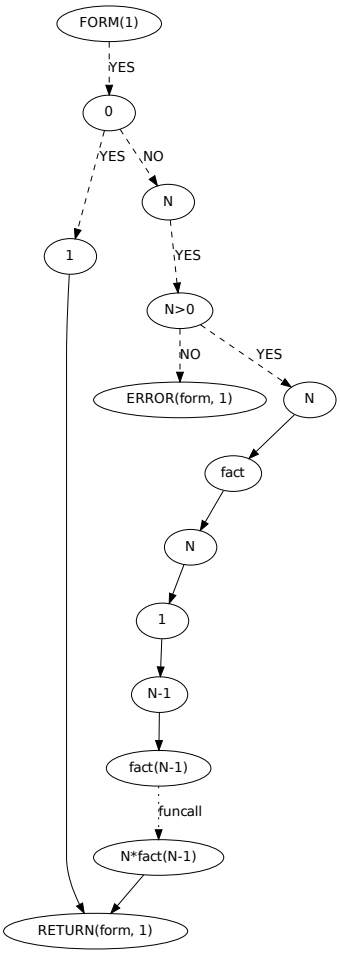


Fig. 15. The CFG for the factorial function: `fact/1` (in Figure 14)

3.10 Dependency Analysis

The Dependency Graph is a labelled directed graph containing the Erlang expressions as its nodes, and data and control dependency among the expressions as its edges. We have introduced three kinds of control dependency edges, one data and one behaviour dependency edge:

- $n_1 \overset{\mathbf{dd}}{\rightarrow} n_2$ denotes that n_2 is directly control dependent from n_1
- $n_1 \overset{\mathbf{resdep}}{\rightarrow} n_2$ (resumption dependent) means that the node n_2 depends on the return of a given function

- $n_1 \overset{\textbf{inhdep}}{\rightarrow} n_2$ represent that node n_2 inherits the dependencies from n_1 based on a function call
- $n_1 \overset{\textbf{datdep}}{\rightarrow} n_2$ represents that n_2 is data dependent from n_1
- $n_1 \overset{\textbf{behdep}}{\rightarrow} n_2$ represents the behaviour dependency

To build the Dependency Graph the first step is to determine which functions are potentially involved in a dependency analysis. We select functions for the initial set and calculate the transitive closure on the call-graph of the functions. Thus we obtain a set of functions, and for these functions we build the Dependency Graph. We have to consider other types of dependencies than function calls as well, which dependencies are message passing and message receiving. When we perform impact analysis, the initial set contains the changed functions.

For the calculated set of functions we build the CFG based on the formal rules described in Appendix C. The CFG is built separately for every function, thus we obtain the intrafunctional CFG for every function. This CFG does not follow the function calls, but denotes the fact of the function call ($\overset{\textbf{call}}{\rightarrow}$), and this information will be used while building the Postdominator Tree (PDT) and the Control Dependency Graph (CDG). This edge is called potential control-flow edge.

Postdominator Tree The Control Dependency Graph is defined with the help of the PostDominator Tree (PDT). A node from the CFG n_1 postdominates n_2, if every execution path from n_2 to *exit* includes n_1, where *exit* is the return node in the CFG of a function. Therefore, we extend the Control-Flow Graphs with a special node, which represents the absolute exit point of the function. We connect the return node and the possible error point of the function with this special node. We build the PDT using the extended CFG. Using the PDT and the extended CFG, we calculate the Immediate Postdominator Tree using the algorithm described in [13]. A node from the CFG n_1 *immediate postdominates* n_2, if and only if n_1 *postdominates* n_2 and $\nexists\ n_3$, $n_2 \neq n_3$, $n_3 \neq n_1$: n_1 *postdominates* n_3 and n_3 *postdominates* n_2.

Immediate Postdominator Tree for function `fact/1` Figure 16 shows the Immediate Postdominator Tree of the factorial function introduced in Figure 14. The $(n_1 \rightarrow n_2)$ relation in the graph means that the node n_1 immediately postdominates the node n_2. The root of the tree is the special exit node `dummy_exit_node`. The entry point of the CFG (`FORM(1)`) is postdominated by this special exit node, which means that the function may exit other than normally.

Control Dependency Graph To determine the control dependencies among the expressions we follow the approach described in [13]. We select those edges from the CFG $(n_1 \rightarrow n_2)$ that are not present in the Immediate Postdominator Tree (n_1 is not postdominated by n_2). With finding the lowest common ancestor of these nodes we can determine dependencies. The nodes on the path starting

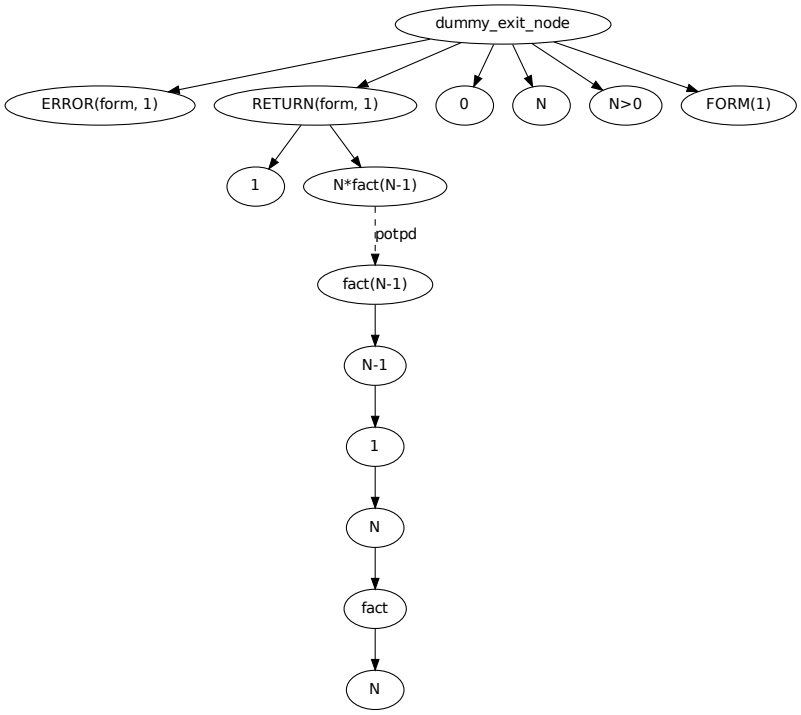


Fig. 16. The postdominator tree for the function `fact/1` (in Figure 14)

from the common ancestor ended in the node n_2 (except the starting node) are control dependent on node n_1.

We want to reduce the cost of rebuilding the Dependency Graph as much as possible. We follow a compositional approach described in [16]. We build Control-Flow Graphs, Postdominator Trees and Control Dependency Graphs separately for each function and compose the CDGs as the last step in the building of the Composed Control Dependency Graph.

Using this approach, the Control Dependency Graphs (CDG) can be maintained separately, and only the composing of the Control Dependency Graphs should be recalculated if something has changed in a subset of functions.

The next level in building the CDG for the entire program is to compose the intrafunctional CDG of the functions. The function calls, send and receive expressions should be examined at this stage. There is a potential dependency among a function application and its postdominators that comes up if there is a potential of not returning from the called function (when the execution of the called function returns abnormally). The dependency among the send and

receive expressions must be also considered. These dependencies can be resolved at the composition stage of the CDGs.

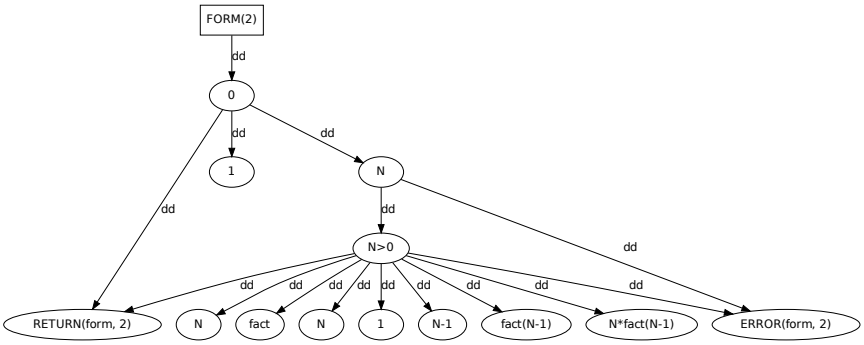


Fig. 17. The Control Dependency Graph for the function `fact/1` (in Figure 14)

Control Dependency Graph for the function `fact/1` Figure 18 shows the Control Dependency Graph for the function `fact/1` in Figure 14. In this graph the potential dependencies were eliminated and were resolved as real dependencies. Let us come up with an example, how the dependencies are determined for the function `fact/1`. There is an edge $(N > 0 \overset{\mathbf{YES}}{\rightarrow} N)$ in the CFG, where `N` does not postdominate the node `N>0`. The lowest common ancestor of these nodes in the Postdominator Tree is the node `dummy_exit_node`, thus nodes on the path from the ancestor to the node `N` (`RETURN(form, 1)`, `N*fact(N-1)`, `fact(N-1)`, ..., `fact` and `N`) are control dependent from `N>0`. Now that we have the intrafunctional CDG of the factorial function, the next step is to resolve the function calls.

There is a function application in the body of the function (as it is a recursive function) to itself. The called function may fail, since it has an error node for the cases if the actual parameter does not satisfy the pattern, or guards. Two new edges are inserted to the CDG

- $(fact(N-1) \overset{\mathbf{inhdep}}{\rightarrow} FORM(1))$, since the application is a recursive call, and
- $(RETURN(form, 1) \overset{\mathbf{resdep}}{\rightarrow} N * fact(N-1))$, since the evaluation of the $(N * fact(N-1))$ depends on the return from the function call.

The old dependency $(fact(N > 0) \overset{\mathbf{inhdep}}{\rightarrow} N * fact(N-1))$ is removed from the CDG, since the resolution of the function call has introduced a new dependency.

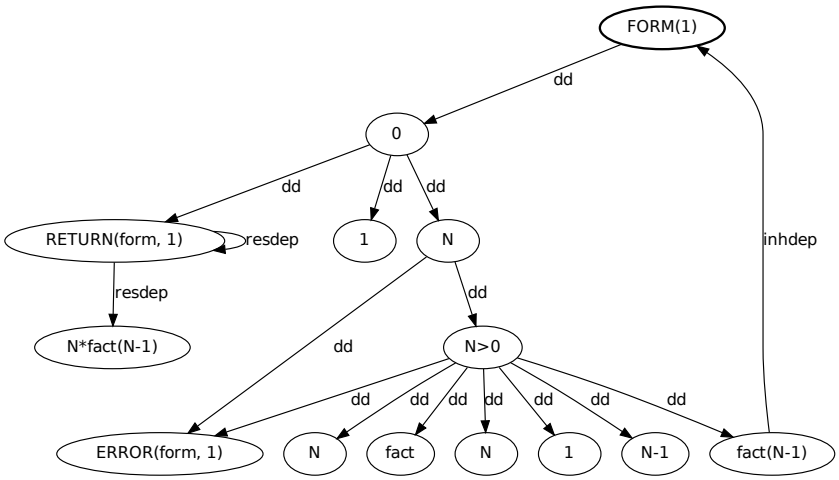


Fig. 18. The resolved Control Dependency Graph for the function `fact/1` (in Figure 14)

Improving the accuracy of the Dependency Graph To reveal real dependencies among the statements of the program, data-flow and data dependency information is also required. The data dependency is calculated from the Data-Flow Graphs of Erlang programs described in Sections 3.1-3.7. The composed CDG is extended with additional data dependencies, thus we obtain the Dependency Graph (DG) presented in Figure 19.

This graph can be extended with additional information like behaviour dependencies from Section 3.8. This will provide information how the behaviour of the function or the entire program is affected, if the data is changed at some statement. With these additional edges we make the DG more accurate. The draft algorithm for creating the Dependency Graph is presented in Figure 20.

Usage of Dependency Graphs The previously described flow and dependency analyses are widely used techniques in compiler optimisations and other static analysis techniques.

We use these Dependency Graphs for static forward slicing [4] of Erlang programs and for finding prallelisable program parts [17].

The slicing is a well known technique to perform static change impact analysis. For slicing we select an expression or a set of expressions and this set will be the slicing criterion. This set can be defined as a result of any change or sequence of changes in the source code. To perform slicing we traverse the Dependency Graph to select the reachable nodes, starting from the criterion set. Performing

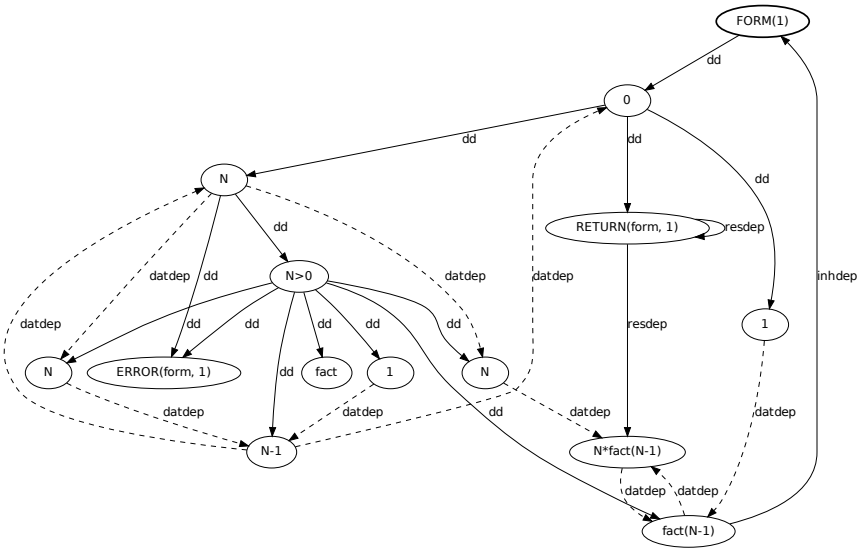


Fig. 19. The Dependency Graph for the function `fact/1` (in Figure 14)

```
calc_dg(SPG)->
  Functions = determine_funs(SPG)
  CFG_List  = lists:map(fun calc_cfg/1, Functions),
  CDG_List  = lists:map(fun calc_cdg/1, CFG_List),
  Comp_CDG  = compose_cdg(CDG_List),
  DCDG      = add_data_dep(CDG)
  BDCDG     = add_behav_and_data_dep(DCDG).
```

Fig. 20. Draft algorithm for creating the Dependency Graph

static forward slicing from these expressions the result can be used to perform impact analysis. As a result of the impact analysis, a subset of the test cases can be selected from the test suite that is possibly affected by the changes. This subset of test cases should be scheduled to run first, because there were changes in the functionalities that the selected test cases are intended to check.

4 Static Analysis in RefactorErl

RefactorErl is a static source code analyser and transformer tool for Erlang. Besides 24 available refactorings, it provides several facilities to support code comprehension tasks and to query information about the source code.

RefactorErl stores the Erlang programs in a Semantic Program Graph (Section 2.1). The lexical level of the graph stores whitespace and comment information, so it can preserve the original layout and comments of the programs during the transformations. The tool has an incremental semantic analyser framework to provide a platform to implement efficient static semantic analyses. RefactorErl stores the SPG in a database, so after a change on the source the stored Semantic Program Graph must be updated. Therefore, the incremental analysis is important, because it has a high cost to reanalyse the whole code base. After each refactoring (or manual change on the source code) only the changed parts are analysed and the necessary information is restored in their context.

The analyser framework is asynchronous and modular: each Erlang form is analysed by a separate Erlang process, and different types of analyses (module, function, variable, record, context, data-flow) are implemented in separate analysers. During the initial analysis the analysers of RefactorErl are executed in a predefined order: an Erlang process is started for every Erlang form, and each process runs the different analysers one by one sequentially. In some cases this ordering is not required (for instance, the record and variable analysers are independent). However some analysers depend on each other, such as the data-flow analyser cannot be performed without the information provided by the variable analyser or the function analyser. The order of the analysers is: context analyser, variable analyser, module analyser, function analyser, record analyser and data-flow analyser. The control-flow analyser runs after the initial loading.

Some analysers can run asynchronously on different forms without any interaction (variables are local to a function clause, so there is no communication between two variable analyser processes), but for analysers using global information synchronisation is required (to create a reference and a definition to a semantic function node must be synchronised).

The data-, control- and behaviour-flow analyses are based on the SPG of RefactorErl. The data and behaviour flow graphs are built during the initial analysis and its edges are added to the SPG. The control-flow graph is built as a separate graph, but it uses the same node identifiers, so the mapping between them is straightforward. We decided to perform the control-flow analysis after the initial loading for efficiency reason. The initial static analysis, without the control-flow analysis, takes almost twelve hours for one and a half million LOC and has 8.5 GB memory footprint. Running control-flow analysis has an extra memory and time cost. The most often used features of RefactorErl could be used without the result of control-flow analysis, so the user can only optionally run this analysis and obtain its result. Currently RefactorErl builds the Dependency Graph only when the slicing application is loaded. In this case the tool monitors the changes made by the performed refactorings on the source code and build the DG of the changed functions. Then it performs the static program slicing to detect the test cases affected by the change on the source code.

Contrarily, the result of data-flow analysis is required for several applications of the tool, therefore the DFG is a part of the semantic level of the SPG. The result of data-flow reaching is used in refactorings: the Introduce Record

refactoring (transform the tuple parameter of a function to a record) calculates reaching on the selected tuple, and transform all reached expressions. The result of the data-flow reaching is available for the users through a Semantic Query Language, and can help in the maintenance task: asking the value of a certain expression (backward reaching) can help to detect and fix failures in the software.

RefactorErl introduces a user level *Semantic Query Language* for Erlang developers to ask information about the source code and to support program comprehension tasks [11, 20].

The language was designed to provide help in the software development process. It uses a formalism close to the Erlang language concepts, thus a developer can easily learn and adopt it.

```
semantic_query    ::= initial_selection ['.' query_sequence]
query_sequence    ::= query ['.' query_sequence]
query             ::= selection | iteration | closure |
                      property_query
initial_selection ::= initial_selector ['[' filter ']']
selection         ::= selector ['[' filter ']']
iteration         ::= '{' query_sequence '}' int ['[' filter ']']
closure           ::= '(' query_sequence ')' int ['[' filter ']'] |
                      '(' query_sequence ')+' ['[' filter ']']
property_query    ::= property ['[' filter ']']
```

Fig. 21. The structure of the semantic queries

The language concept were designed according to the semantic entities of the language, thus it introduces the following *entities*: module, function, variable, record, expression, macro, file. Each entity has a set of *selectors* and *properties*. A selector is a binary relation between two entity types, which selects a set of entities that meet the given requirements. A property is a function, which describes some properties of an entity type. For instance, a module has a selector *funs* to select the function defined in the module, and has a property *name* that defines the name of the module. The result of the query can be *filtered*. A semantic query is a sequence of queries starting with an *initial selector*. There are global initial selectors such as *module*, to select every module form the source code as a starting point of the query, and there are position-based initial selectors (starting with '@') to select the pointed entity in the editor: *@var, @fun, @expr*. It is also possible to iterate queries or calculate the closure of a query.

The list of usable selectors and properties for each entity type can be found in the manual of RefactorErl [2]. Here we present only a few examples of queries.

Call Chain – *@fun.(calls)+* or *@fun.(called_by)+* queries return the forward and backward call chain starting from the pointed function. It is also possible to ask the same result starting from the modules of the analysed program: *mods[name=mymod].funs[name=myfun].(calls)+*

References and Definitions – *@fun.refs, @fun.def, @record.refs*, and similar queries for each entity type return the references or the definition of a given entity. *@fun.dynrefs* returns all the dynamic references to a function.

Original Value of an Expression – *@expr.origin* results in a list of expressions which value can be flown to the pointed expression. The query *@expr.origin[type=atom]* filters out the atoms from the result. In case of debugging, these queries can be useful. For instance, when we get a *badmatch* exception we can find the value of the non matching expression. This query uses the result of the defined data-flow reaching.

Following an Expression – *@expr.reach* lists all of the expressions which value can be a copy of the pointed expression based on the defined data-flow reaching.

Asking dependent nodes – *@expr.dep, @fun.dep* lists all of the expression nodes that depend on the pointed expression or function.

Considering the Erlang code from Figure 7, one can select the variable `Y` from the body of function `cons/0` and query *@expr.origin[type = integer]*. The result contains the integer values that can reach `Y`; in this example this is the integer `2`.

5 Related work

The usage of static analysis techniques was studied in several papers and books. Most of them are closely related to concrete languages or language types.

The book [14] gives a short overview of static analysis techniques and their usage to address different kinds of problems.

Dependency graphs are originally designed and used in compilers to prevent the statement execution in wrong order, i.e. the order that changes the meaning of the program [13]. This book concentrates on high level of optimisations, while our purpose was not the optimisation of Erlang programs, rather to make the information available to the developers. We have utilised some algorithms (e.g. building the postdominator tree) and ideas from this book in our analyses.

Lots of research has been done on the topic of flow analysis. These techniques are mainly used in compiler optimisations, liveness analysis, automatic parallelisation, program slicing, and so on. For instance, Olin Shiver [15] presented a general model for control-flow analysis in Scheme via abstract interpretation of a denotational semantics. The flow analysis was applied to optimisation of higher-order languages such as described in the paper [12]. In case of optimisations, data and control-flow information are calculated simultaneously, but we separate these analyses. Since RefactorErl stores the calculated information in a database, our analyses took more time than an in-memory analysis. Therefore, we decided to calculate only the data-flow information at the initial loading of files and to calculate the control-flow information upon request (e.g. when slicing has to be performed).

It is hard to compare the defined Data-Flow Graphs with flow graphs from other languages. Most of the techniques are based on solving data-flow equations. Those algorithms mainly operate on control-flow graphs, while our algorithm is based on the syntax of Erlang and operates on the extended syntax tree of Erlang programs (i.e. on the Semantic Program Graph of RefactorErl). Using the control-flow graphs they define a set of data-flow equations at the entry and exit point of the basic blocks of the programs and evaluate the flow of data between pred/succ basic blocks [14]. Contrary to this, Erlang is a single assignment language, thus the values of the variables cannot be changed. From this point of view calculating the data-flow reaching in Erlang is a less complex task.

The defined flow graphs and the Dependency Graph can be used in dependency graph based program slicing methods [9]. Most of these algorithms are using compound program or system dependency graphs, which are built from the control- and data-flow graphs of the procedures.

There are static analyser tools for Erlang such as Dialyzer [1]. The goal of this tool is to identify software discrepancies and defects, such as type mismatches, race condition defects, etc. Besides the different goals of Dialyzer and RefactorErl, there is another difference. Dialyzer analyses the Core Erlang code [5] instead of the Erlang source file. Core Erlang is an intermediate representation for Erlang programs, and it has a less complex syntax than Erlang. We decided to analyse the source code and store it in a custom semantic program graph, because RefactorErl aims to preserve the original layout of the unmodified program parts between the refactoring steps.

The refactoring tool Wrangler [10] annotates the Abstract Syntax Tree provided by the Syntax Tools library of Erlang. The Semantic Program Graph representation is more efficient in information retrieval, since instead of syntax tree traversals most of the information about the source code can be gathered by using fixed length queries.

6 Conclusions

The usage of various static analysis methods is getting widespread in different stages of the software development lifecycle. Most of the analysers work on an intermediate source code representation and the abstraction of the representation depends on the target of the static analysis. In this paper we described static semantic knowledge representation about Erlang source code in different forms.

We present a model to represent the lexical, syntactic and semantic information about the Erlang source code, the Semantic Program Graph. Besides the graph, we present a formal model for Erlang programs that is used later to describe formal data-, behaviour- and control-flow rules of Erlang programs. We can build flow graphs based on these rules from the SPG, and further analysis can be performed based on them. We have defined reaching relations on flow graphs. We have described how to build a Dependency Graph using the

flow graphs, and how these graphs can be used for program slicing, or to detect parallelisable components in the source code.

The presented graphs are integrated with the RefactorErl tool, that is a static source code analyser and a refactoring tool for Erlang. We have presented a query language which is applicable to query flow and dependency information about the software for developers.

Acknowledgement

We would like to thank for the related ideas and work of the members of the RefactorErl group.

References

1. *The DIALYZER: a DIscrepancy AnaLYZer for ERlang programs.* http://www.it.uu.se/research/group/hipe/dialyzer.
2. RefactorErl Home Page, 2011. http://plc.inf.elte.hu/erlang/.
3. I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, and M. Tóth. Refactorerl – source code analysis and refactoring in erlang. In *In proceeding of the 12th Symposium on Programming Languages and Software Tools, Tallin, Estonia*, 2011.
4. I. Bozó and M. Tóth. Selecting erlang test cases using impact analysis. In *Proceedings of Symposium on Computer Languages, Implementations and Tools, Kassandra, Halkidiki, Greece*, 2011.
5. R. Carlsson. An introduction to core erlang. In *In Proceedings of the PLI01 Erlang Workshop*, 2001.
6. Ericsson AB. *EDOC – Erlang program documentation generator.* Latest version available online at http://www.erlang.org/documentation/doc-5.4.2.1/lib/edoc-0.1/doc/html/index.html.
7. Ericsson AB. *Erlang Reference Manual.* Latest version available online at http://www.erlang.org/doc/reference_manual/part_frame.html.
8. D. Horpácsi and J. Kőszegi. Static analysis of function calls in erlang – refining the static function call graph with dynamic call information by using data-flow analysis. In *Proceedings of the Central and Eastern European Conference on Software Engineering Techniques*, Debrecen, Hungary, Aug 2011.
9. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PhD thesis, University of Michigan, Ann Arbor, MI*, 1979.
10. H. Li and S. Thompson. Tool support for refactoring functional programs. In *WRT '08: Proceedings of the 2nd Workshop on Refactoring Tools*, pages 1–4, New York, NY, USA, 2008. ACM.
11. L. Lövei, L. Hajós, and M. Tóth. Erlang Semantic Query Language. In Proceedings of the 8th International Conference on Applied Informatics, ICAI 2010, Jan., 2010.
12. J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. In *ACM Transactions on Programming Languages and Systems, volume 20(4), pages 845–868, New York, USA*, 1998.
13. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

14. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999, corrected 2005.
15. O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.
16. J. Stafford. A formal, language-independent, and compositional approach to control dependence analysis. In *PhD thesis, University of Colorado, Boulder, Colorado, USA*, 2000.
17. M. Tóth, I. Bozó, Z. Horváth, and A. Erdődi. Static analysis and refactoring towards erlang multicore programming. In *Pre-proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and CommunicationcEntric Software, PLACES'11, Saarbrcken, Germany*, 2011.
18. M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, and T. Kozsik. Impact analysis of erlang programs using behaviour dependency graphs. In *Central European Functional Programming School. Third Summer School, CEFP 2009. Revised Selected Lectures.*, 2010.
19. M. Tóth, I. Bozó, Z. Horváth, and M. Tejfel. First order flow analysis for Erlang. In *8th Joint Conference on Mathematics and Computer Science, MACS 2010*, 2010.
20. M. Tóth, I. Bozó, J. Kőszegi, and Z. Horváth. Applying the Query Language to support program comprehension. In *Acta Electrotechnica et Informatica, Volume 11, Number 03. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), ISSN 1338-3957 (online)*, pages 3–10, Oct., 2011.

A The Syntax of Erlang Programs

We introduce the Erlang language in this section.

V ::= variables (including the underscore pattern (_))
A ::= atoms
I ::= integers
K ::= $A \mid I \mid$ other constants (e.g. string, float, char)
P ::= $K \mid V \mid \{P,\ldots,P\} \mid$ `[`$P,\ldots,P$`|`P`]`
F ::= A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$`;`
$\vdots$
A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$`.`
E ::= $K \mid V \mid \{E,\ldots,E\} \mid E_{List} \mid$
P `=` $E \mid E \circ E \mid E$ `!` $E \mid \circ E \mid$ `(`E`)` $\mid E$`(`$E,\ldots,E$`)` $\mid$
`case` E `of`
P when E `->` $E,\ldots,E$`;`
$\vdots$
P when E `->` $E,\ldots,E$
`end` $\mid$
`if`
E `->` $E,\ldots,E$`;`
...
E `->` $E,\ldots,E$
`end` $\mid$
`receive`
P when E `->` $E,\ldots,E$`;`
$\vdots$
P when E `->` $E,\ldots,E$
`after`
E `->` $E,\ldots,E$
`end` $\mid E_2$
E_{List} ::= `[`$E,\ldots,E$`|`E`]` $\mid$ `[`E`||`P`<-`$E,\ldots,P$`<-`$E,E,\ldots,E$`]`

Fig. 22. Erlang syntax

The source code of Erlang applications is organised to *modules*. Each module contains a set of *function*, *record* and *macro* definitions and some *attributes* (e.g. module declaration, exported/imported functions, specifications). We introduce the syntax of the Erlang functions in Figures 22 and 23. We do not introduce the syntax of the whole Erlang in these figures; we concentrate on those language elements that are used in the further applied complex semantic analysis such as data-flow or control-flow analyses. During the semantic analysis we consider the Erlang programs as a set of functions. Another simplification of the presented syntax is that the *guard expressions* are represented as regular Erlang expres-

sions, however there are only a few restricted language constructs that can be used as guards. For example, no user defined functions can be used in guards.

```
E2::= try E,...,E of
          P when E -> E,...,E;
          ...
          P when E -> E,...,E
      catch
          P : P when E -> E,...,E;
          ...
          P : P when E -> E,...,E
      after
          E  -> E,...,E
      end |
      catch E |
      begin
          E,...,E
      end |
      fun
          (P,...,P) when E -> E,...,E;
          ...
          (P,...,P) when E -> E,...,E
      end
      fun E/I
```

Fig. 23. Erlang syntax (cont.)

In Figures 22 and 23 we use the following notations:

- V denotes the variables
- K denotes the atoms (A), integers (I) and other constants such as strings, floats, etc.
- P is a pattern that can be any constant, variable or a tuple or list of patterns
- F is a function that has one or more function clauses separated by semicolons. A function clause has a name represented by an atom and has n formal parameters ($n \in N, n \geq 0$) represented by patterns. Optionally, the function clauses have a guard expression after the keyword `when`. The body of the function is a sequence of expressions separated by commas.
- E denotes the expressions. Several kinds of expressions are listed in Figures 22 and 23. An expression can be a constant (K), a variable (V), a tuple ($\{E, ..., E\}$), a list (E_{List}) or can be a compound expression. A list is a sequence of elements optionally followed by the tail of the list in squared brackets and also can be a list comprehension that produces the elements of the list using some generators and filters. A tuple is an ordered list of a fixed number of elements in curly brackets.

The simplest compound expression is the pattern matching expression ($P = E$), which binds a value to a variable. There are infix operators in Erlang ($+, -, and, or$, etc). One of them has a special role: ! is the message passing operator in Erlang. There are also unary operators ($\circ E$), and parenthesis expressions ((E)).
There are different kinds of function applications in Erlang ($E(E, ..., E)$). An application can refer to a fun-expression (lambda expression) or a named function. The named functions can be called by using local or qualified function calls. The latter one refers to the called function with the implementing module name and the called function name ($Mod : Fun(Par_1, ..., Par_n)$).
There are several branching expressions in the language: case, if, receive, try expression, containing some clauses. Their clauses are similar to function clauses. The main difference is that an if clause does not contain any patterns, it only evaluates a guard, and try and receive expression have an optional after clause. Besides the try expression, a simple catch expression was introduced to handle runtime errors.
The begin-end expression is the block expression to group a sequence of expressions into a block.
Finally, like other functional languages, Erlang also introduces unnamed functions as expressions (fun-expression, lambda expression). There are explicit and implicit forms of these expressions. The explicit fun expressions are similar to function definitions and have n clauses ($n \in N, n \geq 1$). The fun expressions begin with the keyword `fun` and are closed with the keyword `end`. The implicit fun expressions refers to named functions.

We note here that during the implementation of the Semantic Program Graph we have extended this syntax description to generate the syntax tree of Erlang modules (details in Section 2.1).

B Data-Flow Rules

We build Data-Flow Graphs for Erlang programs using formal rules. We describe these rules in this section. We use the following notations in the data-flow rules in Figures 24–26:

- $p, p_i \in P$ are patterns,
- $e, e_i \in E$ are Erlang expressions;
- $a \overset{*}{\rightarrow} b$ ($a, b \in P \cup E$, $* \in \{f, c_i, s_i, d\}$) denotes that there is a $*$ type of data-flow edges between nodes a and b

	Expression	Direct Graph Edges
(Variable)	p binding of a variable n occurrence of a variable	$p \overset{\mathbf{f}}{\rightarrow} n$
(Match exp.)	e_0: $p = e$	$e \overset{\mathbf{f}}{\rightarrow} e_0$ $e \overset{\mathbf{f}}{\rightarrow} p$
(Pattern)	p_0: $p_1 = p_2$	$p_0 \overset{\mathbf{f}}{\rightarrow} p_1$ $p_0 \overset{\mathbf{f}}{\rightarrow} p_2$
(Unary op.)	e_0: $\circ\ e_1$	$e_1 \overset{\mathbf{d}}{\rightarrow} e_0$
(Infix op.)	e_0: $e_1 \circ e_2$	$e_1 \overset{\mathbf{d}}{\rightarrow} e_0$ $e_2 \overset{\mathbf{d}}{\rightarrow} e_0$
(Parenthesis)	e_0: (e)	$e \overset{\mathbf{f}}{\rightarrow} e_0$
(Tuple exp.)	e_0: $\{e_1, \ldots, e_n\}$	$e_1 \overset{\mathbf{c_1}}{\rightarrow} e_0, \ldots, e_n \overset{\mathbf{c_n}}{\rightarrow} e_0$
(Tuple pat.)	p_0: $\{p_1, \ldots, p_n\}$	$p_0 \overset{\mathbf{s_1}}{\rightarrow} p_1, \ldots, p_0 \overset{\mathbf{s_n}}{\rightarrow} p_n$
(List exp.)	e_0: $[e_1, \ldots, e_n \vert e_{n+1}]$	$e_1 \overset{\mathbf{c_e}}{\rightarrow} e_0, \ldots, e_n \overset{\mathbf{c_e}}{\rightarrow} e_0$ $e_{n+1} \overset{\mathbf{f}}{\rightarrow} e_0$
(List gen.)	e_0: $[e_1 \vert\vert p \leftarrow e_2]$	$e_1 \overset{\mathbf{c_e}}{\rightarrow} e_0$, $e_2 \overset{\mathbf{s_e}}{\rightarrow} p$
(List pat.)	p_0: $[p_1, \ldots, p_n \vert p_{n+1}]$	$p_0 \overset{\mathbf{s_e}}{\rightarrow} p_1, \ldots, p_0 \overset{\mathbf{s_e}}{\rightarrow} p_n$ $p_0 \overset{\mathbf{f}}{\rightarrow} p_{n+1}$
(BIF 1)	e_0: $\mathsf{hd}(e_1)$	$e_1 \overset{\mathbf{s_e}}{\rightarrow} e_0$
(BIF 2)	e_0: $\mathsf{tl}(e_1)$	$e_1 \overset{\mathbf{f}}{\rightarrow} e_0$
(BIF 3)	I constant, e_0: $\mathsf{element}(I, e_1)$	$e_1 \overset{\mathbf{s_I}}{\rightarrow} e_0$

Fig. 24. Static Data-Flow Graph building rules

Variables Erlang binds a value to a variable in a match expression or in a pattern. This value cannot be changed during the execution of the program. Therefore, the bound value of a variable flows directly to all occurrences of that variable – *Figure 24: (Variable) rule.*

Match Expressions When we bind a value to a variable in a match expression both the value of the pattern expression and the result of the match expression itself are the same as the right hand side expression – *Figure 24: (Match exp.) rule.*

Operators There is no direct data-flow in operator expressions, because an operator does not copy the value of its argument. It evaluates some function using the value on the operands, so the result of the operator expression depends on the value of its operands – *Figure 24: (Unary op.) and (Infix op.) rules.*

Compound expressions and patterns Compound expressions, such as tuples and lists, preserves the value of their elements. For instance, packing some data into a tuple, then forwarding the tuple somewhere in the program (copying its value), and then unpacking the data from the tuple result in the same data. We have to consider only the index of the elements in the compound data structure.

The rules *Figure 24: (Tuple exp.) and (Tuple pat.)* describe that we construct ($\xrightarrow{\mathbf{c_i}}$) the tuple from its elements and we can select ($\xrightarrow{\mathbf{s_i}}$) the elements of the tuple. The tuple constructor and selector edges are indexed by natural numbers ($i \in N$) to denote the position of the elements in the tuple, and the same index in the constructor and in the selector edges represents the same data-flow.

The rule *Figure 24: (List exp.)* describes that we construct the list from some value ($\xrightarrow{\mathbf{c_e}}$) and optionally a list ($\xrightarrow{\mathbf{f}}$) – so, we create it from the head elements and the tail of the list. Like construction, we can select head elements ($\xrightarrow{\mathbf{s_e}}$) and a tail list ($\xrightarrow{\mathbf{f}}$) from a pattern list expression – *Figure 24: (List pat.) rule.* Lists are variable sized data structures and the typical use of them makes precise index-based data-flow calculating useless, so we only distinguish the elements of the list (denoted with the index e) and the tail of the list. In general, the tail of the list contains almost every element of the list, thus we approximate this by adding the flow edge from the tail to the list.

The rule *Figure 24: (List gen.)* shows another way for constructing a list. We select an element (p) from a list (e_2) and push a new element (e_1) to the constructed list (e_0). Most of the time the head expression and the new element depends on the value of the selected element.

BIF – Built in Functions The rules *Figure 24: (BIF 1), (BIF 2) and (BIF 3)* present the selector and constructor data-flow edges based on background knowledge about the given built in functions. The function `hd/1` selects the first element of a list, the function `tl/1` selects the tail of the list and the function `element/2` selects the I^{th} element of a tuple.

	Expressions	Direct Graph Edges
(Case exp.)	e_0: case e of p_1 when $g_1 \rightarrow e_1^1, \ldots, e_{l_1}^1$; ⋮ p_n when $g_n \rightarrow e_1^n, \ldots, e_{l_n}^n$ end	$e \xrightarrow{\mathbf{f}} p_1, \ldots, e \xrightarrow{\mathbf{f}} p_n$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_n}^n \xrightarrow{\mathbf{f}} e_0$
(If exp.)	e_0: if $g_1 \rightarrow e_1^1, \ldots, e_{l_1}^1$; ⋮ $g_k \rightarrow e_1^k, \ldots, e_{l_k}^k$ end	$e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_k}^n \xrightarrow{\mathbf{f}} e_0$
(Fun. call 1)	e_0: m : g$(e_1, \ldots, e_n)$ or g$(e_1, \ldots, e_n)$ m:g/n: g$(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; ⋮ g$(p_1^m, \ldots, p_n^m)$ when $g_m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_1 \xrightarrow{\mathbf{f}} p_1^1, \ldots, e_1 \xrightarrow{\mathbf{f}} p_1^m$ ⋮ $e_n \xrightarrow{\mathbf{f}} p_n^1, \ldots, e_n \xrightarrow{\mathbf{f}} p_n^m$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_m}^m \xrightarrow{\mathbf{f}} e_0$
(Fun. call 2)	e_0: e$_\mathsf{m}$: e$_\mathsf{g}(e_1, \ldots, e_n)$ e_m or e_g not constant or $e_m : e_g/n$ not defined	$e_m \xrightarrow{\mathbf{d}} e_0$, $e_g \xrightarrow{\mathbf{d}} e_0$ $e_1 \xrightarrow{\mathbf{d}} e_0, \ldots, e_n \xrightarrow{\mathbf{d}} e_0$

Fig. 25. Static Data-Flow Graph building rules (cont.)

Branching expressions Conditional expressions branch the control based on pattern matching (case expression) or guard evaluation (if expression). The result of such an expression is always the result of the last expression of the branch evaluated at runtime. Since potentially any branch can be evaluated, the value of each last expression can flow to the case/if expression. Besides this, the result of the head expression of the case expression is matched to the patterns of each branch, thus it has data-flow among them – *Figure 25: (Case exp.) and (If exp.) rules.*

Function calls There are two different function call rules – *Figure 25: (Fun. call 1) and (Fun. call 2).* The difference between the two rules is that in the former case we can unambiguously identify the called function and its body, but in the latter we cannot detect the called function body (because the AST of the implementing module is not available or the module or the function name is dynamic).

	Expressions	Direct graph Edges
(Try exp.)	e_0: try $e_1, \ldots, e_k$of p_1 when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \rightarrow$ $e_1^n, \ldots, e_{l_n}^n$ catch p_{n+1} when $g_{n+1} \rightarrow$ $e_1^{n+1}, \ldots, e_{l_{n+1}}^{n+1}$; $\vdots$ p_m when $g_m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$ after $e_{m+1} \rightarrow$ $e_1^{m+1}, \ldots, e_{l_{m+1}}^{m+1}$ end	$e \xrightarrow{\mathbf{f}} p_1, \ldots, e \xrightarrow{\mathbf{f}} p_n$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_n}^n \xrightarrow{\mathbf{f}} e_0$ $e_{l_{n+1}}^{n+1} \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_m}^m \xrightarrow{\mathbf{f}} e_0$
(Catch exp.)	e_0: catch e	$e \xrightarrow{\mathbf{f}} e_0$
(Begin-end)	e_0: begin $e_1, \ldots, e_n$ end	$e_n \xrightarrow{\mathbf{f}} e_0$
(Send exp.)	e_0: e_1 ! e_2 e': receive p_1 when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \rightarrow$ $e_1^n, \ldots, e_{l_n}^n$ after $e \rightarrow e_1, \ldots, e_s$ end	$e_2 \xrightarrow{\mathbf{f}} e_0$ $e_2 \xrightarrow{\mathbf{f}} p_1, \ldots, e_2 \xrightarrow{\mathbf{f}} p_n$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e', \ldots, e_{l_n}^n \xrightarrow{\mathbf{f}} e'$ $e_s \xrightarrow{\mathbf{f}} e'$

Fig. 26. Static Data-Flow Graph building rules (cont.)

If the body of the called function is available in our representation, we can preform an interprocedural data-flow reaching; otherwise we can only denote the dependency from the called function and the parameters of the call (denoted by ($\xrightarrow{\mathbf{d}}$) edges). In the former case we can find the actual parameters of the called function and add flow edges form the formal parameters to the corresponding actual parameter of each function clause. The result of a function call is the

result of the last expression of the executed function body, therefore we add flow edges from the last expressions to the function call expression.

Error handling expressions The try expression rule (*Figure 26: (Try exp.)*) is similar to the case rule. The head of the try expression contains more expressions and the result of the last expression is matched to the patterns of the try, but it does not match the patterns of the catch clauses. These are evaluated when a runtime exception occurs and the exception matches them. Thus, there is no data-flow among the last expressions and the patterns of the catch clauses. The result of the try expression is the result of the evaluated clause, so the values of the last expressions (including the values of the catch clauses) flow to the try expression. The result of the after clause is simply omitted, it does not flow anywhere.

The catch expression rule (*Figure 26: (Catch exp.)*) describes that the result of its body flows to the catch. In case of a runtime exception , the result of the catch is the error report of the exception.

Message sending and receiving The message sending operator (!) differs from the other infix operators. Its return value is the value of its right hand side expression, so the value of the sent massage – *Figure 26: (Send exp.) rule.* The message flows to the addressed process and tries to match one of its receive expressions, so the message flows to the patterns of the corresponding receive expressions. A naive data-flow algorithm should add a flow edge to the patterns of each receive expression to represent the potential data-flow. This could result in a huge amount of edges in the graph. Instead of this, we try to calculate the corresponding receive expressions and connect them with the sent messages (for details see Section 3.7).

The receive expression is similar to other branching expressions, so its return value is the last expression of the evaluated clause. Therefore, the value of the last expression of each clause flows to the receive expression.

Implicit and Explicit Fun Expressions (Lambda Expressions) The rules of fun expressions (*Figure 27: (Fun. exp. 1) and (Fun. exp. 2)*) express similar parameter value and result copying as functions and function calls, but in the most of the cases identifying them is not straightforward. The fun expressions are defined in the body of functions and they can spread among functions as data, so data-flow analysis is required to identify the definitions of fun expressions.

If it is possible to identify the definition of the explicit fun expression – *(Fun. exp 1) rule* – we link the actual parameter of the call and the corresponding formal parameter of each fun expression clause with a flow edge, and add a flow edge from the last expression of each function body to the call representing the return value.

If data-flow reaching detects that the defining expression of the fun expression is an implicit fun expression, we have to find the definition of the referred function. Similar to the *(Fun. call 2.) rule* (Figure 26), if the AST is not available, we have to add the dependency edges to the Data-Flow Graph

	Expressions	Direct Graph Edges
(Fun exp. 1)	e: $\mathsf{fun}(p_1^1,\ldots,p_n^1)$ when g_1 $\rightarrow$ $e_1^1,\ldots,e_{l_1}^1$; $\vdots$ $(p_1^m,\ldots,p_n^m)$ when g_m $\rightarrow$ $e_1^m,\ldots,e_{l_m}^m$ e_0: $e(e_1,\ldots,e_n)$ e can be calculated by data-flow analysis	$e_1 \stackrel{\mathbf{f}}{\rightarrow} p_1^1,\ldots,e_1 \stackrel{\mathbf{f}}{\rightarrow} p_1^m$ $\vdots$ $e_n \stackrel{\mathbf{f}}{\rightarrow} p_n^1,\ldots,e_n \stackrel{\mathbf{f}}{\rightarrow} p_n^m$ $e_{l_1}^1 \stackrel{\mathbf{f}}{\rightarrow} e_0,\ldots,e_{l_m}^m \stackrel{\mathbf{f}}{\rightarrow} e_0$
(Fun exp. 2.)	m:g/n: $\mathsf{g}(p_1^1,\ldots,p_n^1)$ when g_1 $\rightarrow$ $e_1^1,\ldots,e_{l_1}^1$; $\vdots$ $\mathsf{g}(p_1^m,\ldots,p_n^m)$ when g_m $\rightarrow$ $e_1^m,\ldots,e_{l_m}^m$. e: fun $m:g/n$ or fun g/n e_0: $e(e_1,\ldots,e_n)$ e can be calculated by data-flow analysis	$e_1 \stackrel{\mathbf{f}}{\rightarrow} p_1^1,\ldots,e_1 \stackrel{\mathbf{f}}{\rightarrow} p_1^m$ $\vdots$ $e_n \stackrel{\mathbf{f}}{\rightarrow} p_n^1,\ldots,e_n \stackrel{\mathbf{f}}{\rightarrow} p_n^m$ $e_{l_1}^1 \stackrel{\mathbf{f}}{\rightarrow} e_0,\ldots,e_{l_m}^m \stackrel{\mathbf{f}}{\rightarrow} e_0$
(Fun. exp 3)	e_0: $e(e_1,\ldots,e_n)$ e cannot be detected by data-flow reaching or e is m:g/n or g/n by data-flow reaching but m:g/n or g/n not defined	$e \stackrel{\mathbf{d}}{\rightarrow} e_0$ $e_1 \stackrel{\mathbf{d}}{\rightarrow} e_0,\ldots,e_n \stackrel{\mathbf{d}}{\rightarrow} e_0$

Fig. 27. Static Data-Flow Graph building rules (cont.)

(*Figure 27: (Fun. exp. 3)*), otherwise we add the flow edges among the parameters and the return values (*Figure 27: (Fun. exp. 2)*).

The rule *Figure 27: (Fun. exp. 3)* describes the dependency edges when no information can be calculated about the referred function or fun expression using data-flow analysis.

Dynamic Function Calls The rules of dynamic function calls (Figure 28) are also based on the reuse of data-flow analysis. When it is possible to detect the referred functions by data-flow analysis we link the actual parameters to the formal parameters and the return value to the function call by flow edges. The *(Dyn. call. 1)* rule describes the MFA-s (qualified function calls where the name of the module and/or the name of the function are not atoms) when the module

	Expressions	Direct Graph Edges
(Dyn. call 1)	e_0: $e_1 : e_2(e_3, \ldots, e_{n+2})$ $e_1 : e_2/n$ is m:g/n by data-flow reaching m:g/n: $\mathsf{g}(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{g}(p_1^m, \ldots, p_n^m)$ when $g_m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_3 \xrightarrow{\mathbf{f}} p_1^1, \ldots, e_3 \xrightarrow{\mathbf{f}} p_1^m$ $\vdots$ $e_{n+2} \xrightarrow{\mathbf{f}} p_n^1, \ldots, e_{n+2} \xrightarrow{\mathbf{f}} p_n^m$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_m}^m \xrightarrow{\mathbf{f}} e_0$
(Dyn. call 2)	e_0: $\mathsf{apply}(e_1, e_2, e_3)$ e_1 is m, e_2 is g, e_3 is $[e_4, \ldots e_{n+3}]$ by data-flow reaching m:g/n: $\mathsf{g}(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{g}(p_1^m, \ldots, p_n^m)$ when $g_m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_4 \xrightarrow{\mathbf{f}} p_1^1, \ldots, e_4 \xrightarrow{\mathbf{f}} p_1^m$ $\vdots$ $e_{n+3} \xrightarrow{\mathbf{f}} p_n^1, \ldots, e_{n+3} \xrightarrow{\mathbf{f}} p_n^m$ $e_{l_1}^1 \xrightarrow{\mathbf{f}} e_0, \ldots, e_{l_m}^m \xrightarrow{\mathbf{f}} e_0$
(Dyn. call 3)	e_0: $e_1 : e_2(e_3, \ldots, e_{n+2})$ $e_1, \ldots e_{n+2}$ cannot be detected by data-flow reaching	$e_1 \xrightarrow{\mathbf{d}} e_0, \ldots, e_{n+2} \xrightarrow{\mathbf{d}} e_0$
(Dyn. call 4)	e_0: $\mathsf{apply}(e_1, e_2, e_3)$ e_1, e_2, e_3 cannot be detected by data-flow reaching	$e_1 \xrightarrow{\mathbf{d}} e_0, e_2 \xrightarrow{\mathbf{d}} e_0, e_3 \xrightarrow{\mathbf{d}} e_0$

Fig. 28. Static Data-Flow Graph building rules (cont.)

name and the function name is statically detectable. The *(Dyn. call. 2)* rule describes that in case of an apply call the parameter list of the actual call also has to be detected. We have to calculate the arity of the function and it is also necessary to link them to the formal parameters of the referred function.

If one of the necessary information is not reachable, we only have to add the data dependency edges – *(Dyn. call. 3) and (Dyn. call. 4) rules.*

C Control-Flow Rules

The Control-Flow Graph (CFG) represents all the possible execution/evaluation paths of the program that can be chosen for every possible input. The CFG is a language dependent representation of the program as it is based on the semantics of the language.

We have defined the control-flow rules for Erlang programming language based on its semantics. The language has strict evaluation, which means that before evaluating a compound expression its subexpressions have to be evaluated. In every case the subexpressions are evaluated in left to right order. The defined control-flow rules are compositional, thus the graph can be composed from the previously computed subgraphs. We use the SPG of RefactorErl and use the same identifiers for the vertices in the CFG. We extend the set of nodes with additional dummy vertices for joining branches, error nodes, etc. The rules are defined in Figures 29–32.

The notation in the figures are: $e, e_i \in E$ is an expression, $g, g_j \in E$ is a guard expression, $p, p_k \in P$ is a pattern and $f/n \in F$ stands for function. The $e'_0 \in E$ is a dummy node in the control flow graph, which represents the entry point of the compound expression and joining dummy nodes (ret) to represent the return value for the conditional branching expressions. The relation $\rightarrow$ represents the control-flow between the nodes. The edges that have no labels represent sequences, and edges with labels represent:

- conditional branching and pattern matching with ($\overset{\mathbf{yes}}{\rightarrow}$), ($\overset{\mathbf{no}}{\rightarrow}$) edges
- returning to a previous expression ($\overset{\mathbf{ret}}{\rightarrow}$),
- function calls/applications with ($\overset{\mathbf{call}}{\rightarrow}$),
- receive expression with ($\overset{\mathbf{rec}}{\rightarrow}$),
- send expression with ($\overset{\mathbf{send}}{\rightarrow}$).

The relations ($\overset{\mathbf{call}}{\rightarrow}$), ($\overset{\mathbf{rec}}{\rightarrow}$), ($\overset{\mathbf{send}}{\rightarrow}$) represent special relations, which indicate a possible dependency between the nodes of different functions (for details, see Section 3.10). In the rest of this section we describe the formal rules for different expression and discuss the rules in more detail.

Unary operator There are only a few unary operators in Erlang, like `+`, `-`, `bnot`, `not`, etc. In the case of the unary operators (*Figure 29: (Unary op.)*), first the subexpression is evaluated ($e'_0 \rightarrow e_1$), then the unary operator is applied on the evaluated subexpression ($e_1 \rightarrow e_0$).

Left associative infix expression The rule in *Figure 29: (Left assoc. op.)* describes the control-flow in left associative expressions. The language is strict, the subexpressions are evaluated first from left to right order ($e_1 \rightarrow e_2$) and then the operator ($e_2 \rightarrow e_0$). The subexpressions are evaluated from left to right order.

If there is a sequence of left associative operators with the same precedence, the sequence of operators are evaluated from left to right order. First the first

	Expressions	Control-Flow Edges
(Unary op.)	e_0: $\circ\ e_1$	$e'_0 \to e_1$, $e_1 \to e_0$
(Left assoc. op.)	e_0: $e_1 \circ_1 e_2 \circ_2 ...$ $\circ_{n-2}\ e_{n-1}\ \circ_{n-1}\ e_n$	$e'_0 \to e_1$, $e_1 \to e_2$, $e_2 \to \circ_1$, $\circ_1 \to e_3 \ \ldots\ e_n \to \circ_{n-1}$, $\circ_{n-1} \to e_0$
(Right assoc. op.)	e_0: $e_1 \circ_1 e_2 \circ_2\ \ldots$ $\circ_{n-2}\ e_{n-1}\ \circ_{n-1}\ e_n$	$e'_0 \to e_1$, $e_1 \to e_2\ ...\ e_{n-1} \to e_n$, $e_n \to \circ_{n-1}$, $\circ_{n-1} \to \circ_{n-2}, ..., \circ_2 \to \circ_1$, $\circ_1 \to e_0$
(Comp. infix op.)	e_0: $e_1 \circ e_2$	$e'_0 \to e_1$, $e_1 \to e_2$, $e_2 \to e_0$
(Andalso op.)	e_0: $e_1 \circ e_2$	$e'_0 \to e_1$, $e_1 \overset{\textbf{yes}}{\to} e_2$, $e_1 \overset{\textbf{no}}{\to} e_0$, $e_2 \to e_0$
(Orelse op.)	e_0: $e_1 \circ e_2$	$e'_0 \to e_1$, $e_1 \overset{\textbf{no}}{\to} e_2$, $e_1 \overset{\textbf{yes}}{\to} e_0$, $e_2 \to e_0$
(Send op.)	e_0: $e_1\ !\ e_2$	$e'_0 \to e_1$, $e_1 \to e_2$, $e_2 \overset{\textbf{send}}{\to} e_0$
(Parenthesis)	e_0: (e_1)	$e'_0 \to e_1$, $e_1 \to e_0$
(Tuple exp.)	e_0: $\{e_1, \ldots, e_n\}$	$e'_0 \to e_1$, $e_1 \to e_2,\ \ldots,\ e_{n-1} \to e_n$, $e_n \to e_0$
(List exp.)	e_0: $[e_1, \ldots, e_n \vert e_{n+1}]$	$e'_0 \to e_1$, $e_1 \to e_2,\ \ldots,\ e_n \to e_{n+1}$, $e_{n+1} \to e_0$

Fig. 29. Control-Flow Graph building rules

two subexpressions ($e_1 \to e_2$) of the first operator are evaluated and the operator is applied to these values ($e_2 \to \circ_1$), then the previous result and the result of the third subexpression ($\circ_1 \to e_3$) using the second operator ($e_3 \to \circ_3$) and so on. The left associative infix expressions are: `/`, `*`, `div`, `rem`, `band`, `and`, etc.

Right associative infix expression The rule in *Figure 29: (Right assoc. op.)* describes the control-flow in right associative expressions. The language is strict, the subexpressions are evaluated first and next the operator. The subexpressions are evaluated from left to right order.

If there is a sequence of right associative operators with the same precedence, the sequence of subexpressions is evaluated($e_1 \to e_2$, $e_2 \to e_3$, ..., $e_{n-1} \to e_n$) and then the operators are evaluated from right to left order. First the result of

	Expressions	Control-Flow Edges
(List gen. 1)	e_0: $[e\|\|p_1\texttt{<-}e_1, \ldots, p_n\texttt{<-}e_n]$	$e'_0 \to e_1$, $e_i \to p_i$, $p_i \overset{\textbf{no}}{\to} e_i$, $p_i \overset{\textbf{yes}}{\to} e_{i+1}$, $e_i \overset{\textbf{ret}}{\to} e_{i-1}$, $e \to e_1$, $(i \in [1, ..., n], e_{n+1} = e)$
(List gen. 2)	e_0: $[e\|\|p_1\texttt{<-}e_1, f_{(1,0)}, \ldots, f_{(1,m_1)}$ $\vdots$ $p_n\texttt{<-}e_n, f_{(n,0)}, \ldots, f_{(1,m_n)}]$	$e'_0 \to e_1$, $e_i \to p_i$, $p_i \overset{\textbf{no}}{\to} e_i$, $e_i \overset{\textbf{ret}}{\to} e_{i-1}$, $p_i \overset{\textbf{yes}}{\to} f_{(i,0)}$, $f_{(i,j-1)} \overset{\textbf{yes}}{\to} f_{(i,j)}$, $f_{(i,m_i)} \overset{\textbf{yes}}{\to} e_{i+1}$, $f_{(i,0)} \overset{\textbf{no}}{\to} e_i$, $f_{(i,j)} \overset{\textbf{no}}{\to} e_i$, $e \to e_1$, $(i \in [1, ..., n], j \in [1, ..., m_i], n, m_i \in N$ $e_{n+1} = e)$
(List gen. 3)	e_0: $[e\|\|f_{(0,0)}, \ldots, f_{(0,m_0)},$ $p_1\texttt{<-}e_1, \ldots, p_n\texttt{<-}e_n]$	$e_i \to p_i$, $p_i \overset{\textbf{no}}{\to} e_i$, $p_i \overset{\textbf{yes}}{\to} e_{i+1}$, $e_i \overset{\textbf{ret}}{\to} e_{i-1}$, $e'_0 \to f_{(0,0)}$, $f_{(0,j-1)} \overset{\textbf{yes}}{\to} f_{(0,j)}$, $f_{(0,m_0)} \overset{\textbf{yes}}{\to} e_1$, $f_{(0,0)} \overset{\textbf{no}}{\to} e_0$, $f_{(0,j)} \overset{\textbf{no}}{\to} e_0$, $e \to e_1$, $(i \in [1, ..., n], j \in [1, ..., m_0], n, m_0 \in N$ $e_{n+1} = e)$

Fig. 30. Control-Flow Graph building rules (cont.)

the last two subexpressions using the last operator is evaluated, next the previous result and the third subexpression and so on ($\circ_{n-1} \to \circ_{n-2}$, ..., $\circ_2 \to \circ_1$). The right associative operators are: `++`, `--`, `=`, `!`, etc.

Comparative infix expression The comparative infix expressions are neither left nor right associative. The rule in *Figure 29: (Comp. infix op.)* describes the control-flow of these expressions. The two subexpressions are evaluated first from left to right order, then the comparison is evaluated. There are comparison expressions like: `<`, `>`, `=<`, `>=`, etc.

Short-circuit expressions (andalso, orelse) The evaluation of the language is strict, but there are two short-circuit infix expressions. The first of them is expression `andalso` (*Figure 29: (Andalso op.)*), which evaluates its left argument first. If it evaluates to `true` the control is given to the right argument ($e_1 \overset{\textbf{yes}}{\to} e_2$), otherwise it returns with the result `false` and lets the right expression non-evaluated ($e_1 \overset{\textbf{no}}{\to} e_0$).

The second short-circuit expression is `orelse` (*Figure 29: (Orelse op.)*). It evaluates the left argument and if it evaluates to `false`, it continues with evalu-

ating the right argument ($e_1 \overset{\textbf{no}}{\rightarrow} e_2$), otherwise returns with result `true` and lets the right argument non-evaluated ($e_1 \overset{\textbf{yes}}{\rightarrow} e_0$).

Send operator (!) The control-flow of the send operator is described in *Figure 29: (Send op.)*. The send operator is right associative, but we describe its control-flow separately. The message sending has side effect and may affect the control-flow of other processes. By analysing the sent messages we can improve the accuracy of our analysis, thus where the send expression is detected the edge is labelled with `send` tag. The evaluation of the send expression is analogous to the right associative expressions. First the subexpressions are evaluated from left to right order, then the send expressions are evaluated from right to left order. The return value of the send expression is the sent value.

Parenthesis With parentheses we can modify the precedence of the expressions. The control-flow rules for this expression are described in *Figure 29: (Parenthesis)*. We first evaluate the expression in the parentheses and then the parent expression gets the control.

Tuple expression The `n-tuples` are to couple coherent data with fixed size of elements, such as messages, etc. The control-flow of the `n-tuples` are defined in *Figure 29: (Tuple exp.)*. The elements of the tuple expression are evaluated from left to right order and then it resumes the control to the parent expression, which constructs the tuple.

List expression The control-flow of list expressions (*Figure 29: (List exp.)*) is similar to the `n-tuples`. The elements of the list are evaluated and then the control is passed to the parent expression.

List comprehension The control-flow rules for building the CFG of the list comprehension is defined in *Figure 30: (List gen. 1), (List gen. 2) and (List gen. 3) rules*. The list comprehension is composed of the head expression, which is an arbitrary expression, and a list of qualifiers. A qualifier is a list of either a generator or a filter expression. These three rules cover every possible list comprehension constructs and can be combined. The first rule *(List gen. 1)* describes the control-flow between the generator expressions, the second rule *(List gen. 2)* describes the control-flow among generator and filter expressions and among filter expressions, the third rule *(List gen. 3)* describes the case when the first element in a qualifier list is a filter expression.

The qualifiers are evaluated in left to right order. If the qualifier is a generator, its list expression is evaluated first (for example: $(e_0 \rightarrow e_1)$). It tries to match the values against its pattern $(e_1 \rightarrow p_1)$. If it succeeds, then continues with the next qualifier, which can be either a generator or a filter expression. If none of the values match the pattern, then it resumes the control to its preceding expression (for example the $e_1 \overset{\textbf{ret}}{\rightarrow} e_0$).

If the qualifier is a filter expression, it is evaluated. If it evaluates to `true` the control is resumed by the next qualifier, which can be either a generator or a filter expression. For example:

– to the next filter expression ($f_{(i,j)} \overset{\textbf{yes}}{\rightarrow} f_{(i,j+1)}$) or
– to the list expression of the next generator ($f_{(i,j)} \overset{\textbf{yes}}{\rightarrow} e_{i+1}$)

If the filter expression evaluates to `false` the control is resumed to the closest generator situated to its left. For example: $f_{(i,j)} \overset{\textbf{no}}{\rightarrow} e_i$).

If the end of the qualifier list is reached and even the last qualifier meets the requirements, the head expression is evaluated (($f_{(n,m_n)} \overset{\textbf{yes}}{\rightarrow} e$) or ($p_n \overset{\textbf{yes}}{\rightarrow} e$)) and again the control is handed to the first qualifier ($e \rightarrow e_1$).

Function In Erlang the function may have several function clauses as the pattern matching and the guard expressions play a special role in control-flow and branching of possible execution paths. The rules for constructing the control-flow graph of the functions is described in *Figure 31:(Function)*. The actual parameters are matched against the formal parameters/patterns and guard expressions sequentially. If the pattern matches, then the guard expression is evaluated. If the guard evaluates to `true` this clause will be chosen for evaluation. If either the pattern matching fails or the guard expression evaluates to `false` the control flows to the next function clause. The expressions in the body of the function are evaluated sequentially and subexpressions are evaluated according to the rules described in this section. The return value of the function is the last evaluated expression from its body.

Function call The rules of the control-flow in a function application is defined in *Figure 31: (Fun. call)*. First the expression that defines the module and name of the function is evaluated ($e'_0 \rightarrow e_f$), then the evaluation of the actual parameters follows. The actual parameters are evaluated from left to right order (($e_1 \rightarrow e_2$), ..., ($e_{n-1} \rightarrow e_n$)). Then the evaluation should pass to the called function. Therefore, the ($e_n \overset{\textbf{call}}{\rightarrow} e_0$) edge indicates an interfunctional control-flow, which should be considered during the building of the control dependency graph.

Case expression The rules for building the control-flow for the case expression is described in *Figure 32: (Case exp.)*. First the head expression is evaluated, then the return value of the evaluated head expression is matched against the patterns. The control flow of the pattern matching is analogous to the one described at the *(Function)* rule. The branch of the first matching pattern and optional guard that evaluates to `true` will be evaluated. If the pattern does not match or the guard is evaluated to `false` the next pattern is examined. The return value of the case expression is the value of the last expression of the evaluated branch.

Receive expression The rules of control-flow of the receive expression are described in *Figure 32: (Receive exp)*. The receive expression tries to remove a message from the message queue and matches it against the patterns and guards similarly as the case expression. The execution of the process may hang until it receives an appropriate message, thus the receiving is marked with the special label `rec` in the control-flow.

	Expressions	Control-Flow Edges
(Function)	$\mathsf{f/n}$: $\mathsf{f}(p_1^1, \ldots, p_n^1)$ when $g^1 \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{f}(p_1^m, \ldots, p_n^m)$ when $g^m \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$f/n \rightarrow p_1^1$, $\{p_1^1, \ldots, p_n^1\} \overset{\textbf{yes}}{\rightarrow} g^1$, $\{p_1^1, \ldots, p_n^1\} \overset{\textbf{no}}{\rightarrow} \{p_1^2, \ldots, p_n^2\}$, $\vdots$ $\{p_1^{m-1}, \ldots, p_n^{m-1}\} \overset{\textbf{yes}}{\rightarrow} g^{m-1}$, $\{p_1^{m-1}, \ldots, p_n^{m-1}\} \overset{\textbf{no}}{\rightarrow} \{p_1^m, \ldots, p_n^m\}$, $\{p_1^m, \ldots, p_n^m\} \overset{\textbf{yes}}{\rightarrow} g^m$, $\{p_1^m, \ldots, p_n^m\} \overset{\textbf{no}}{\rightarrow} error$, $g^1 \overset{\textbf{yes}}{\rightarrow} e_1^1$, $g^1 \overset{\textbf{no}}{\rightarrow} \{p_1^2, \ldots, p_n^2\}$, $\vdots$ $g^{m-1} \overset{\textbf{yes}}{\rightarrow} e_1^{m-1}$, $g^{m-1} \overset{\textbf{no}}{\rightarrow} \{p_1^m, \ldots, p_n^m\}$, $g^m \overset{\textbf{yes}}{\rightarrow} e_1^m$, $g^m \overset{\textbf{no}}{\rightarrow} error$, $e_1^1 \rightarrow e_2^1, \ \ldots, \ e_{l_1-1}^1 \rightarrow e_{l_1}^1$, $\vdots$ $e_1^m \rightarrow e_2^m, \ \ldots, \ e_{l_m-1}^m \rightarrow e_{l_m}^m$, $e_{l_1}^1 \rightarrow ret\ f/n$, $\vdots$ $e_{l_m}^m \rightarrow ret\ f/n$,
(Fun. call)	e_0: $\mathsf{e_f}(e_1, \ldots, e_n)$	$e_0' \rightarrow e_f$, $e_f \rightarrow e_1$, $e_1 \rightarrow e_2, \ \ldots, \ e_{n-1} \rightarrow e_n$, $e_n \overset{\textbf{call}}{\rightarrow} e_0$,

Fig. 31. Control-Flow Graph building rules (cont.)

	Expressions	Control-Flow Edges
(Case exp.)	e_0: case e of p_1 when $g_1 \to e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \to e_1^n, \ldots, e_{l_n}^n$ end	$e_0' \overset{\mathbf{e}}{\to}$, $e \to p_1$, $p_1 \overset{\mathbf{yes}}{\to} g_1$, $p_1 \overset{\mathbf{no}}{\to} p_2$, $\vdots$ $p_{n_1} \overset{\mathbf{yes}}{\to} g_{n-1}$, $p_{n-1} \overset{\mathbf{no}}{\to} p_n$, $p_n \overset{\mathbf{yes}}{\to} g_n$, $p_n \overset{\mathbf{no}}{\to} error$, $g_1 \overset{\mathbf{yes}}{\to} e_1^1$, $g_1 \overset{\mathbf{no}}{\to} p_2$, $\vdots$ $g_{n-1} \overset{\mathbf{yes}}{\to} e_1^{n-1}$, $g_{n-1} \overset{\mathbf{no}}{\to} p_n$, $g_n \overset{\mathbf{yes}}{\to} e_1^n$, $g_n \overset{\mathbf{no}}{\to} error$, $e_1^1 \to e_2^1, \ \ldots, \ e_{l_1-1}^1 \to e_{l_1}^1$, $\vdots$ $e_1^n \to e_2^n, \ \ldots, \ e_{l_n-1}^n \to e_{l_n}^n$, $e_{l_1}^1 \to ret\ case$, $\vdots$ $e_{l_n}^n \to ret\ case$, $ret\ case \to e_0$
(Receive exp.)	e_0: receive p_1 when $g_1 \to e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \to e_1^n, \ldots, e_{l_n}^n$ end	$e_0' \overset{\mathbf{rec}}{\to} p_1$, $\vdots$ Similarly as at rule (Case exp.) $\vdots$ $e_{l_1}^1 \to ret\ receive$, $\vdots$ $e_{l_n}^n \to ret\ receive$, $ret\ receive \to e_0$

Fig. 32. Control-Flow Graph building rules (cont.)

1^{ST} ORDER FLOW ANALYSIS FOR ERLANG

MELINDA TÓTH, ISTVÁN BOZÓ, ZOLTÁN HORVÁTH, MÁTÉ TEJFEL

Abstract. Flow analysis supports many software maintenance tasks. In this paper we define 1^{st} order data-flow analysis for a higher-order dynamically typed functional programming language, Erlang, as an improvement of the 0^{th} order analysis. We present the differences between the data-flow graph building rules of the two analysis and introduce a $1st$ data-flow relation that is suitable to calculate indirect flow relation on the refined graph.

1. Introduction

Related studies have already examined a number of flow analysis techniques for various programming languages. Some of these analysis techniques are heavily used during software development as well as in software maintenance. They are mainly used, for example, in optimisation, in impact analysis, in program slicing, in debugging, in liveness analysis and so on.

Testing is one of the most expensive parts of the software development process. Tool support for reducing the number of necessary test cases could decrease testing costs. Our goal is to develop algorithms that perform program slicing and give the possibility of deriving a subset of test cases that have to be re-tested after a particular change on the source code. This paper focuses on defining an accurate data-flow graph for a higher-order dynamically typed functional programming language, Erlang. The data-flow graph carries information about data dependency, which is essential for dependency graph based program slicing.

In this paper we present the rules of static data-flow analysis according to the evaluation of Erlang programs [5] and we also show the static flow analysis algorithms we have developed for Erlang. This form of analysis is conservative in a way that it includes every possible data-flow that may occur in the program at runtime. This kind of analysis can be more accurate if context information [2] have also been considered in the analysis process. This context information may be, for instance, the calling context, the message passing context, and so on. To define the analysis in a context sensitive way, we present the generalisation of the so-called 0^{th} order flow analysis to

2010 *Mathematics Subject Classification.* 68Q99.

Key words and phrases. Erlang, static analysis, data-flow.

Supported by TECH 08 A2-SZOMIN08, ELTE IKKK and Ericsson Hungary.

1^{st} order flow analysis. Here we note that the 0^{th} order data-flow analysis has already been studied in previous papers [4, 7].

RefactorErl is a semantic program graph based refactoring tool for Erlang. Its semantic graph consists of an AST (abstract syntax tree) supplemented with static semantic information provided by static semantic analysers. The semantic information may be, for example, the binding structure of the variables, record usage, function call graph, and so on. A 0^{th} order static data-flow analyser is included in the RefactorErl's [8] analyser framework. Section 3 illustrates the inadequacy of 0^{th} order data-flow analysis. Therefore it will be replaced by the generalised and improved 1^{st} order data-flow analyser.

Our main goal is to perform impact analysis of Erlang programs and to measure the impact of any change caused by a code refactoring. Based on the 1^{st} order Data-Flow Graphs (built upon the presented rules in Section 4) the performed program slicing [6] will result in more accurate slices. Consequently, this results in a more precise measurement of the effect of any change.

The rest of this paper is structured as follows: Section 2 describes data-flow analysis for Erlang and its extension with type heuristic; Section 3 shows an example that motivated us to improve the existing model; Section 4 introduces the first order flow rules and relation; Section 5 shows how we could generalise our rules; Section 6 presents related work. Finally, Section 7 concludes the paper and indicates some future work.

2. Flow analysis for Erlang

2.1. **The Erlang syntax.** In the following we use the Erlang syntax defined in [7]. A subset of the syntax is shown on Figure 1, where A denotes the Erlang atoms, V marks the variables, P denotes patterns, E denotes expressions and F may stand for functions.

$P ::= V \mid \{P,\ldots,P\} \mid \ldots$
$E ::= V \mid \{E,\ldots,E\} \mid P \texttt{ = } E \mid E\texttt{(}E,\ldots,E\texttt{)} \mid \ldots$
$F ::= A\texttt{(}P,\ldots,P\texttt{)}$ when E `->` $E,\ldots,E\texttt{;}$
$\vdots$
$A\texttt{(}P,\ldots,P\texttt{)}$ when E `->` $E,\ldots,E\texttt{.}$

Figure 1. The used Erlang syntax subset

2.2. **Data-flow rules.** The 0^{th} order data-flow analysis has been studied in previous papers [4, 7]. Based on the syntax defined in Figure 1 we have created compositional data-flow rules. These rules describe data-flow relations among expressions. For instance, Figure 2 contains some simple rules:

- The *Variable* rule describes that when we bind a value to a variable, each reference to this variable will represent the same value, thus the value of the binding expression flows into the references.
- The *Match Exp.* rule means that the value of the right hand side of a match expression flows into its left hand side pattern and into the match expression itself.
- The *Tuple exp.* and *Tuple pat.* rules describe Erlang specific data construction and selection. We can pack ($\xrightarrow{c_i}$) n element to a tuple (e_0). The packed values can flow into another expression, and afterwards we can unpack ($\xrightarrow{s_i}$) the elements from the tuple (p_0). In that case we have to deduce the fact that the value of e_1 flows into p_1. We handle the list expressions similarly as the tuple expressions, just the index of the elements can not be calculated, thus it denoted with e.

	Expressions	Graph edges
(Variable)	p is a binding n is a usage of the same variable	$p \xrightarrow{f} n$
(Match exp.)	e_0: $p = e$	$e \xrightarrow{f} e_0$ $e \xrightarrow{f} p$
(Tuple exp.)	e_0: $\{e_1, \ldots, e_n\}$	$e_1 \xrightarrow{c_1} e_0, \ldots, e_n \xrightarrow{c_n} e_0$
(Tuple pat.)	p_0: $\{p_1, \ldots, p_n\}$	$p_0 \xrightarrow{s_1} p_1, \ldots, p_0 \xrightarrow{s_n} p_n$

FIGURE 2. Example data-flow rules

The problem with the 0^{th} order flow analysis is the lack of the ability of distinguishing different function calls (for more details see Section 3). Figure 3 shows the data-flow rule for an ordinary function call where the name and arity of the called function are given. When a function is called, the values of the actual parameters (e_i) flow into the formal parameters (p_i). The return value of the function is the value of the last expression of its body, consequently, the value of the last expression ($e^i_{l_i}$) flows back into the function call (e_0) according to the evaluation rules of Erlang programs presented in [5].

2.3. **Type heuristic.** The *Function call* rule is rather imprecise, since the actual parameters of the function call are linked to the formal parameters of each function clause. However, in most of the cases the actual parameters will not match the patterns of every function clause. Therefore, we refine the data-flow graph in a way that we only link the actual and formal parameters that could be matched. In spite of the fact that there are many different kinds of mismatches, we only focus on the simple cases in which

	Expressions	Graph edges
(Fun. call)	e_0: $\mathtt{g}(e_1, \ldots, e_n)$ $\mathtt{g/n}$: $\mathtt{g}(p_1^1, \ldots, p_n^1)$ when g_1 $\rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathtt{g}(p_1^m, \ldots, p_n^m)$ when g_m $\rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$ $e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$ $\vdots$ $e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$

FIGURE 3. Zeroth order function call rule

it is guaranteed the type of the actual and formal parameters can not be unified.

For example, lets consider function *sel* which can take two different types of arguments: a one-element list and a one-element tuple (Figure 4). Obviously, when we call *sel* from the body of *test_list*, the first function clause will match (`sel([A]) -> A`) and the value of `[3]` never flows into the pattern `{B}`. Therefore, when it is possible to determine the types of the formal and actual parameters, we only link parameters that are of the same type (see Figure 5).

Here we note that this refinement is applicable because of the fact that the refactorings implemented in RefactorErl are type-preserving transformations. If we would like to apply our data-flow graph for impact analysis of a non-type-preserving change on the source code, then we should take the original data-flow graph.

2.4. **Data-flow reaching.**

Definition 1: Zeroth order data-flow relation. The data-flow relation $\overset{0f}{\rightsquigarrow}$ is defined as the minimal relation that satisfies the following rules:

(reflexive) $n \overset{0f}{\rightsquigarrow} n$

(c-s-rule) $$\frac{n_1 \xrightarrow{c_i} n_2,\ n_2 \overset{0f}{\rightsquigarrow} n_3,\ n_3 \xrightarrow{s_i} n_4}{n_1 \overset{0f}{\rightsquigarrow} n_4}$$

(f-rule) $$\frac{n_1 \xrightarrow{f} n_2}{n_1 \overset{0f}{\rightsquigarrow} n_2}$$

(transitive) $$\frac{n_1 \overset{0f}{\rightsquigarrow} n_2,\ n_2 \overset{0f}{\rightsquigarrow} n_3}{n_1 \overset{0f}{\rightsquigarrow} n_3}$$

We use this data-flow relation to follow the impact of the change of the source code. If expression n_1 is in a data-flow relation with expression n_2 ($n_1 \overset{0f}{\rightsquigarrow} n_2$), then changing the value of n_1 affects the value of n_2. If n_2 is in an expression of a test function, then we have to rerun this test. If $\nexists n_i$ in the body of a test function where $n_1 \overset{0f}{\rightsquigarrow} n_i$, then we can ignore to rerun this test case.

3. Motivating example

Figure 4 shows an example illustrating some shortcomings of the 0^{th} order data-flow analysis. The example contains a selector function (*sel*) that can extract the element either from a one-element tuple or from a one-element list. In addition, a function g using this selector function is included. Two possible calls of the selector function and an application of function g are tested in *test_list*, *test_tuple* and *test_g*. In the case if a type-preserving transformation affecting the match expression `Z = {A}` was executed, then using the original version of the 0^{th} order data-flow analysis all the three test cases would be selected for re-testing. The reason is that the 0^{th} order analysis cannot distinguish the different calling points, so the changes caused by the transformation flow back to every caller of the function *sel*, including calls in *test_list* and *test_tuple* as well.

```
sel([A]) ->                 test_list() ->
    A;                          sel([3]) == 3.
sel({B}) ->
    B.                      test_tuple() ->
                                sel({4}) == 4.
g(A) ->
    Z = {A},                test_g() ->
    sel(Z).                     g(5) == 5.
```

Figure 4. Example: Erlang functions

If we use the refined version of the 0^{th} order analysis in the same example, only tests *test_tuple* and *test_g* will be selected for re-testing, since by using type heuristic we can distinguish the clauses of the function *sel*. Changes caused by the described transformation flow only to the calling points passing a one-element tuple as parameter.

The data-flow graph of this case is illustrated on Figure 5. The vertices of the graph represent the Erlang (sub)expressions from Figure 4. Each vertex contains an identifier ($\$I$), the text of the subexpression and the text of the containing topmost expression. The labelled edges of the graph represent the direct data-flow among expressions corresponding to the mentioned data-flow rules. Data belonging to the expression `Z = {A}` (represented by node \$7, \$8 and \$9) flows into node \$14 (belonging to *test_tuple*) and \$16 (belonging to *test_g*) and do not flow into node \$6 (belonging to *test_list*).

Note that this refined analysis is still imprecise, since the expected result is the selection of only one test case, namely, the function *test_g*. The expected result can be achieved by applying 1^{st} order data-flow analysis, which is described in Section 4.

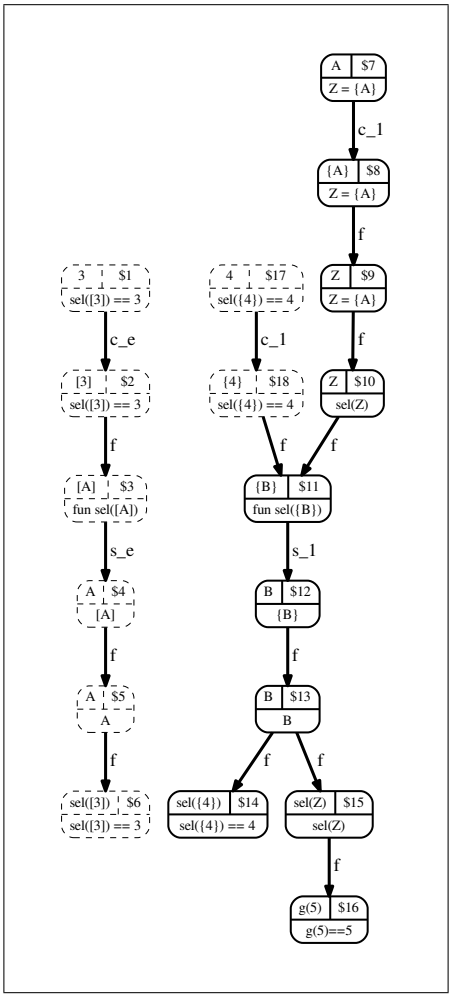


FIGURE 5. 0^{th} order graph

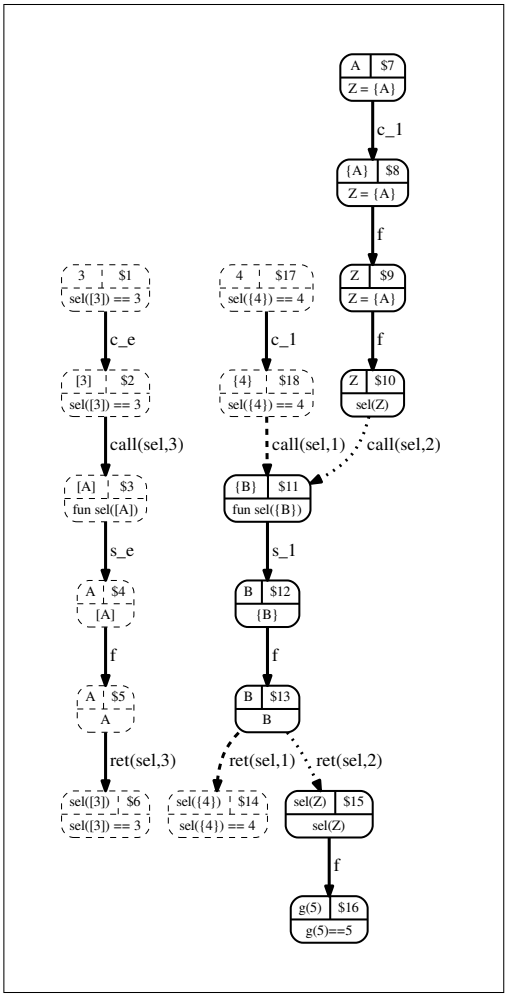


FIGURE 6. 1^{st} order graph

4. First order data-flow analysis

4.1. **Extended first order rules.** To achieve first order analysis, the 0^{th} order data-flow rules have to be extended with some context information. Our motivating example indicates that we should denote a differentiation among calls to a function. Therefore, we store more precise information on the entry and exit points of a function rather than considering only the flow of the parameters.

Our first order analysis introduces new data-flow relations: $\xrightarrow{call(g)}$ for entering and $\xrightarrow{ret(g)}$ for leaving the function g. The context information is added to the edge as an index: $\xrightarrow{call(g,i)}$ means the i^{th} call of the function, and $\xrightarrow{ret(g,i)}$

means the return point of the i^{th} call. Figure 6 presents the draft data-flow graph including the new edges and Figure 7 shows the compositional data-flow rule for a function call.

	Expressions	Graph edges
(Fun. call)	e_0: $\mathsf{g}(e_1, \ldots, e_n)$ g/n: $\mathsf{f}(p_1^1, \ldots, p_n^1)$ when g_1 $\rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $\mathsf{f}(p_1^m, \ldots, p_n^m)$ when g_m $\rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_{l_1}^1 \overset{ret(g,i)}{\longrightarrow} e_0, \ldots, e_{l_m}^m \overset{ret(g,i)}{\longrightarrow} e_0$ $e_1 \overset{call(g,i)}{\longrightarrow} p_1^1, \ldots, e_1 \overset{call(g,i)}{\longrightarrow} p_1^m$ $\vdots$ $e_n \overset{call(g,i)}{\longrightarrow} p_n^1, \ldots, e_n \overset{call(g,i)}{\longrightarrow} p_n^m$

FIGURE 7. First order function call rule

4.2. **Extended first order data-flow relation.** The presented data-flow rules describe direct flow information among Erlang expressions. Since the data could flow from node n_1 to n_2, and from n_2 to n_3, we are curious whether the value of n_1 flows indirectly to n_3. The 1^{st} order data-flow relation returns those nodes in the graph where a value of a given node can flow through a sequence of the data-flow edges.

We derive the first order data-flow relation from the zeroth order data-flow relation considering the following:

- $\overset{0f'}{\rightsquigarrow}$ denotes the zeroth order data-flow relation calculated on the data-flow graph defined by the extended data-flow rules. The zeroth order flow relation operates on $\overset{f}{\longrightarrow}$ edges, but the first order function call rule replaces the corresponding $\overset{f}{\longrightarrow}$ edges with $\overset{call}{\longrightarrow}$ and $\overset{ret}{\longrightarrow}$ edges, therefore the zeroth order reaching cannot use them and this results in a smaller intrafunctional $\overset{0f'}{\rightsquigarrow}$ relation.
- $\overset{1f[\mu]}{\rightsquigarrow}$ denotes the first order data-flow relation.
- In the first order relation ($\overset{1f[\mu]}{\rightsquigarrow}$) μ is a list of call ($\overset{call(g,i)}{\longrightarrow}$) and return ($\overset{ret(h,j)}{\longrightarrow}$) points. We have to record the names and the indices of the called functions, because later we have to find the corresponding exit points.
- Each node (n_i) that is reachable in the extended representation with the 0^{th} order data-flow relation is reachable by the first order relation (0^{th} *flow rule*).
- Similarly to the 0^{th} order relation, if a data constructor packs ($\overset{c_i}{\longrightarrow}$) the node n_1 into n_2 and the value of n_2 flows (with a first order

flow) into the node n_3 and another data constructor unpacks ($\xrightarrow{s_i}$) the value into n_4, then the value of n_1 flows into n_4 (1^{st} *c-s rule*).

- The call ($\overset{call(g,i)}{\longrightarrow}$) and the return ($\overset{ret(h,j)}{\longrightarrow}$) edges behave similarly to the flow $\xrightarrow{f}$ edges, so the data flows through them (*call rule* and *return rule*).
- The data can flow through any function call (*call concat. rule*).
- If the value of the node n_1 flows into the node n_2 through the return value of a function call and the value of n_2 flows into the node n_3 through the return value of another function call, then the value of n_1 transitively flows into the node n_3 (*return concat. rule*).
- If we enter the function through the edge $\overset{call(g,i)}{\longrightarrow}$, then we have to leave the function through the $\overset{ret(g,i)}{\longrightarrow}$ edge (*reduce rule*) and leaving the function body through an $\overset{ret(g,j)}{\longrightarrow}$ $(j \neq i)$ edge is not allowed (*Lemma* 3.).

In Definition 2 we the following notations: μ denotes a list; $hd(\mu)$ results the head (first) element of a list; $last(\mu)$ stands for the last element of a list; $\mu + +\rho$ denotes the concatenation of list μ and list ρ; μ_n denotes the n^{th} element of list μ.

Definition 2: First order data-flow relation. The data-flow relation ($\overset{1f}{\rightsquigarrow}$) is the minimal relation that satisfies the following rules:

(0^{th} flow rule)
$$\frac{n_1 \overset{0f'}{\rightsquigarrow} n_2}{n_1 \overset{1f[]}{\rightsquigarrow} n_2}$$

(1^{st} c-s rule)
$$\frac{n_1 \xrightarrow{c_i} n_2,\ n_2 \overset{1f[\mu]}{\rightsquigarrow} n_3,\ n_3 \xrightarrow{s_i} n_4}{n_1 \overset{1f[\mu]}{\rightsquigarrow} n_4}$$

(call rule)
$$\frac{n_1 \overset{call(g,i)}{\longrightarrow} n_2}{n_1 \overset{1f[call_{(g,i)}]}{\rightsquigarrow} n_2}$$

(return rule)
$$\frac{n_1 \overset{ret(h,j)}{\longrightarrow} n_2}{n_1 \overset{1f[ret_{(h,j)}]}{\rightsquigarrow} n_2}$$

(call concat. rule)
$$\frac{n_1 \overset{1f[\mu]}{\rightsquigarrow} n_2,\ n_2 \overset{1f[\rho]}{\rightsquigarrow} n_3}{n_1 \overset{1f[\mu++\rho]}{\rightsquigarrow} n_3} \quad if\ (\exists f\ \exists i:\ (hd(\rho) = call_{(g,i)}))\ or\ \rho = []$$

(return concat. rule)
$$\frac{n_1 \overset{1f[\mu]}{\rightsquigarrow} n_2,\ n_2 \overset{1f[ret_{(h,j)}|\rho]}{\rightsquigarrow} n_3}{n_1 \overset{1f[\mu++[ret_{(h,j)}|\rho]]}{\rightsquigarrow} n_3} \quad if\ (\exists f\ \exists i:\ (last(\mu) = ret_{(g,i)}))\ or\ \mu = []$$

(reduce rule)
$$\frac{n_1 \overset{1f[\mu++[call_{(h,i)}]]}{\rightsquigarrow} n_2,\ n_2 \overset{1f[ret_{(h,i)}]}{\rightsquigarrow} n_3}{n_1 \overset{1f[\mu]}{\rightsquigarrow} n_3}$$

The next few lines describes some properties of the defined relation:

Property 1. The defined first order relation is reflexive: $\exists \mu : n \overset{1f[\mu]}{\rightsquigarrow} n$

Proof.

(0^{th} flow rule)
$$\frac{n \overset{0f'}{\rightsquigarrow} n\ \text{(reflexive)}}{n \overset{1f[]}{\rightsquigarrow} n}$$

Property 2. If $n_1 \overset{1f[\mu]}{\rightsquigarrow} n_2$ and $\mu_n = call_{(g,i)}$, then $\mu_m \neq ret_{(h,j)}$ $(m \geq n)$.

Proof. Applying structural induction on $n_1 \overset{1f[\mu]}{\rightsquigarrow} n_2$.

Property 3. $\overset{1f[\mu]}{\rightsquigarrow}$ is neither transitive, symmetrical nor anti-symmetrical.

Proof. Counterexamples are based on the Property 2. and the property of the 0^{th} order data-flow relation presented in [7].

4.3. **Example reaching.** Figure 6 shows the draft first order data-flow graph of our motivating example in Section 3. Using the first order reaching relation we get a more precise result. Starting the reaching from the node n_7 (marked with identifier \$7 on Figure 6) results that the node n_{14} (marked with identifier \$14) is not affected by the change of the value of the expression. Applying Definition 2. on the example graph gives the following result:

(0^{th} flow rule)
$$\frac{n_8 \overset{0f'}{\rightsquigarrow} n_{10}}{n_8 \overset{1f[]}{\rightsquigarrow} n_{10}}$$

(call rule)
$$\frac{n_{10} \overset{call(sel,2)}{\longrightarrow} n_{11}}{n_{10} \overset{1f[call_{(sel,2)}]}{\rightsquigarrow} n_{11}}$$

(call concat. rule)
$$\frac{n_8 \overset{1f[]}{\rightsquigarrow} n_{10},\ n_{10} \overset{1f[call_{(sel,2)}]}{\rightsquigarrow} n_{11}}{n_8 \overset{1f[call_{(sel,2)}]}{\rightsquigarrow} n_{11}}$$

(1^{st} c-s rule)
$$\frac{n_7 \xrightarrow{c_1} n_8,\ n_8 \overset{1f[call_{(sel,2)}]}{\rightsquigarrow} n_{11},\ n_{11} \xrightarrow{s_1} n_{12}}{n_7 \overset{1f[call_{(sel,2)}]}{\rightsquigarrow} n_{12}}$$

(0^{th} flow rule) $$\frac{n_{12} \overset{0f'}{\rightsquigarrow} n_{13}}{n_{12} \overset{1f[]}{\rightsquigarrow} n_{13}}$$

(return rule) $$\frac{n_{13} \overset{ret(sel,2)}{\longrightarrow} n_{15}}{n_{13} \overset{1f[ret_{(sel,2)}]}{\rightsquigarrow} n_{15}}$$

(return concat. rule) $$\frac{n_{12} \overset{1f[]}{\rightsquigarrow} n_{13},\ n_{13} \overset{1f[ret_{(sel,2)}]}{\rightsquigarrow} n_{15}}{n_{12} \overset{1f[ret_{(sel,2)}]}{\rightsquigarrow} n_{15}}$$

(reduce rule) $$\frac{n_7 \overset{1f[call_{(sel,2)}]}{\rightsquigarrow} n_{12},\ n_{12} \overset{1f[ret_{(sel,2)}]}{\rightsquigarrow} n_{15}}{\mathbf{n_7} \overset{\mathbf{1f[]}}{\rightsquigarrow} \mathbf{n_{15}}}$$

5. Further improvements

The discussed first order flow analysis is a good refinement of the zeroth order data-flow analysis. Since Erlang is a higher-order language and we can pass functions as arguments to a function, there are some cases where the context information stored by the first order analysis is not enough. Let's consider the source code from Figure 8. If we call function $func$ from function h, then the `Fun(Data)` application calls function *pear*, but if we call function $func$ from function g, then the `Fun(Data)` application calls function *apple*.

```
func(Fun, Data)->
    Fun(Data).
```

```
h()->
    f(fun pear/1, [pear]).

g()->
    f(fun apple/1, [apple]).
```

Figure 8. Higher order functions

If we detect that the `Fun(Data)` application may call both function *apple* and function *pear* and we create the ordinary $\overset{call(pear,i)}{\longrightarrow}$ and $\overset{call(apple,j)}{\longrightarrow}$ first order flow edges, then during calculating reaching from the body of h we will reach both function *apple* and function *pear*. A solution to that problem could be to store two depth calling context: $\overset{call((func,1);(pear,i))}{\longrightarrow}$ and $\overset{call((func,2);(apple,j))}{\longrightarrow}$.

This second order flow analysis could be generalised as well and so on: we can generalise the data-flow algorithm to an n^{th} order flow analysis ($\overset{call((func_1,i_1);\ \ldots\ ;(func_n,i_n))}{\longrightarrow}$).

6. Related work

Many researches have been done in the topic of data-flow analysis. These techniques are mainly used in compiler optimisations, liveness analysis, automatic parallelisation, program slicing, and so on. For instance, Olin Shiver [2] presented a general model for control-flow analysis in Scheme via abstract interpretation of a denotational semantics. That flow analysis was applied to optimisation of higher-order languages such as it is described in paper [3].

The flow graphs are also the bases of the dependency graph based program slicing methods [6], where the compound program or system dependency graphs are built from the control- and data-flow graphs of the procedures.

It is hard to compare the defined data-flow graphs with flow graphs from other languages. Most of the techniques are based on solving data-flow equations. Those algorithms mainly operate on control-flow graphs, while our algorithm is based on the syntax of Erlang and operates on the extended syntax tree of Erlang programs (i.e. on the Semantic Program Graph of RefactorErl). Using the control-flow graphs they define a set of data-flow equations at the entry and exit point of the basic blocs of the programs and evaluate the flow of data between pred/succ basic blocks [11].

7. Conclusions

The main goal of our project is to perform program slicing to measure the impact of a change made on the source code. We want to achieve this by using dependency graph based static program analysis, where a proper data-flow analysis results in a smaller program slice. In previous papers we studied zeroth order data-flow analysis [4, 7], and in this paper we presented the generalisation of that analysis by storing context information about the function calls. We discussed the importance and relevance of the first order flow analysis in Section 3.

The presented higher order analysis could reduce the size of resulting program slice, but each step made to build a more accurate graph results in a more complex and time-consuming analysis. In the future we want to focus on analysing the cost of the analysis against the reduced test case number.

In spite of the fact that we aim to use the data-flow analysis for building dependency graphs and performing program slicing, their usage is more widespread. The detected data-flow information could help to improve refactorings [4, 9] and also to support program comprehension [10].

References

[1] Erlang Homepage, *https://www.erlang.org*

[2] Shivers, O.: Control-Flow Analysis of Higher-Order Languages, PhD thesis, Carnegie Mellon University, Pittsburgh, USA, May, 1991

[3] Michael Ashley, J. and Kent Dybvig, R.: A practical and flexible flow analysis for higher-order languages, ACM Transactions on Programming Languages and Systems, volume 20(4), pages 845–868, New York, USA, July, 1998
[4] Lövei, L.:, Automated module interface upgrade, Erlang '09: Proceedings of the 8th ACM SIGPLAN workshop on Erlang, ISBN 978-1-60558-507-9, pages 11–22, Edinburgh, Scotland, September, 2009
[5] Fredlund, L.-A.: A Framework for Reasoning about ERLANG code, PhD thesis, Royal Institute of Technology, Stockholm, Sweden, August, 2001
[6] Horwitz, S. ,Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs ACM Transactions on Programming Languages and Systems, volume 12(1), pages 26–60, New York, USA, January, 1990
[7] Tóth, M., Bozó, I., Horváth, Z., Lövei, L., Tejfel, M. and Kozsik, T.: Impact analysis of Erlang programs using behaviour dependency graphs, Central European Functional Programming School, Revised Selected Lectures, Komárno, Slovakia, June, 2009
[8] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg A., Nagy, T., Tóth, M. and Király, R.: Modeling semantic knowledge in Erlang for refactoring, Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, volume 54(2009) Sp. Issue, Studia Universitatis Babe-Bolyai, Series Informatica, Cluj-Napoca, Romania, July, 2009
[9] Lövei, L., Horváth, Z., Kozsik, T. and Király, R.: Introducing Records by Refactoring, Proceedings of the 6th ACM SIGPLAN Erlang Workshop, pages 18–28, Freiburg, Germany, October, 2007
[10] Tóth, M., Bozó, I. and Horváth, Z.: Applying the Query Language to support program comprehension, In proceedings of the International Scientific Conference on Computer Science and Engineering, pages 52–59, Stara Lubovna, Slovakia, September, 2010
[11] Nielson, F., Nielson, H. R. and Hankin C.: Principles of Program Analysis, Springer, 1999.

Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers
E-mail address: {toth_m,bozo_i,hz,matej}@inf.elte.hu

B. függelék

Adatfüggőség elemzés

A függelékben szereplő cikk [TBH+10b] végleges változatának publikációs adatai:

- Melinda Tóth, István Bozó, Zoltán Horváth, László Lövei, Máté Tejfel, and Tamás Kozsik. Impact analysis of erlang programs using behaviour dependency graphs. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, pages 372–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

Impact analysis of Erlang programs using behaviour dependency graphs *

Melinda Tóth, István Bozó, Zoltn Horváth, László Lövei,
Máté Tejfel, Tamás Kozsik

Eötvös Loránd University, Budapest, Hungary
{toth_m,bozo_i,hz,lovei,matej,kto}@inf.elte.hu

Abstract. During the lifetime of a software product certain changes could be performed on its source code. After those changes a regression test should be performed, which is the most expensive part of the software development cycle. This paper focuses on programs written in a dynamic functional programming language Erlang, and discusses a mechanism that could select those test cases, which are affected by a change, i.e. altering the program on some point may have impact on the result/behaviour of those test cases. In the result of that analysis it is possible to reduce the number of necessary test cases, and after modifying the source code, just a subset of the test cases should be retested. The discussed approach introduces a behaviour dependency graph for Erlang programs to represent the dependencies in the source code. The impact of a change can be calculated by traversing the graph.

1 Introduction

Changes often happen in a software lifetime. These changes can be done manually by a programmer or using a refactoring tool. The phase "refactoring" [3] introduces a meaning preserving source code transformation, thus you change the structure of a program without altering its external behaviour. Refactoring could be done manually by a programmer or using a refactoring tool. The former case is tedious and error prone, the latter is safer and faster. A refactoring tool guarantees that the transformation does not change the meaning of the program and all the necessary changes will happen. However refactoring in Erlang [2] is not straightforward. The language is dynamically typed, so the syntactic and static semantic information sometimes may be not enough to guarantee a meaning preserving transformation, and the programmer want to test the behaviour of the transformed program. Since testing is a very expensive part of the software development process, we want to help the programmers to reduce the number of test cases which should be performed after a transformation or a sequence of transformations. Therefore we try to find the affected parts in the source code by analyzing the spread of the impact of performed changes. Assume we have

* Supported by TECH_08_A2-SZOMIN08, ELTE IKKK, and Ericsson Hungary

found the affected code parts, then only a subset of test cases should be retested, those which are affected by the change.

To find the affected parts we have to propagate the change of some data, therefore we introduce a behaviour dependency graph. During the generation of this graph we use static syntactic and semantic information based on the semantic program graph of the RefactorErl.

RefactorErl [6, 5] is a refactoring tool for Erlang. To represent the source code the tool uses its own semantic program graph model, which contains lexical, syntactic and semantic information about the loaded Erlang programs. The graph is based on the AST. RefactorErl uses its own layout preserving parser to generate the syntax tree, then different semantic analyzers (function, variable, record, etc) extend the syntax tree to semantic program graph. The constructed program graph provides good interface for further source code analysis.

The rest of this paper is structured as follows. In Section 2 a motivating example is given. Section 3 introduces the used model of Erlang programs. Then the Section 4 introduces the behaviour dependency graph, the method of constructing the behaviour dependency edges and the way of retrieving dependency information from this graph. Section 5 presents related work, and Section 6 concludes the paper and discusses future work.

2 Motivating example

For the sake of simplicity we demonstrate our mechanism in a more general example. We do not transform the source code by a refactoring and analyze its impact, rather we modify an element of a list (Figure 1 and 2) and then we estimate the impact of the data change to determine the test cases which should be retested.

Consider the following example (Figure 1), where we define the `tag_add/1` recursive function, which transforms the elements of the given list to a tagged tuple. We expect from this function, that it does not alter the length of the given list. The `test_tag_add/2` function is intended to describe this property of the `tag_add/2` function. The `len/1` function calculates and returns the length of the given list. In our example the `tag_list/0` function calls the `tag_add/2` function with a list containing two integers: `[1,2]`.

Assume, that the programmer modify the first element of the list `[1,2]` in the body of `tag_list/0`. This value flows into the variable `H1` (in `tag_add/2`) through the list construct. The result of the `tag_add/2` function is a list which spine does not depend on the value of `H1` variable, however the elements of this result list depend on the value of `H1`. Therefore, we have to detect whether the elements of the resulted list are used elsewhere in the code. It can be used those point in the program where the `tag_add/2` function is called. The `test_tag_add/2` function calls the `tag_add/2` function, thus it uses its return value, so the elements of the resulted list may be used. The `test_tag_add/2` function passes the result of the `tag_add(L1, Tag)` function call as an argument for the `len/1` function and calculates the length of the resulted list with

```
tag_add([], _Tag) ->
    [];
tag_add([H1|T1], Tag)->
    [{Tag, H1} | tag_add(T1, Tag)].

tag_list() ->
    tag_add([1,2], integer).

test_tag_add(L1, Tag)->
    len(L1) == len(tag_add(L1, Tag)).

len([]) ->
    0;
len([_H | T]) ->
    1 + len(T).
```

Fig. 1. The definition of `tag_add/2`

`len/1`, but during the calculation `len/1` does not use the value of the elements coming form its argument (in the body of `len/1` the values of the elements (`_H`) of the parameter list are not used). Therefore the result of the `len(L1)` and `len(tag_add(L1))` function calls do not depend on the values of the elements coming from the `L1` list and the return value of the `tag_add(L1)` function call. We can see, that changing any element of the list `[1,2]` under writing the program does not have any impact on the result of these function calls, thus does not have any impact on the return value and on the behaviour of the `test_tag_add/2` function, so we must not retest it.

However the elements of the list depend on the operation (a tuple constructor), in this case the structure of the list is independent of this operation. The change of one list element does not always has impact on the structure of the list or the context of the list usage.

In the second part of the example we give the definition of a similar function, but the behaviour of these functions are different. On Figure 2 we define the `tag_filter/2` recursive function, which filters the elements of the list with a given tag `Tag`. This function selects a sublist of the given list, it may throw out element from the list. Thus changing the elements of the parameter list may affect the spine of the resulted list. This function may keep or decrease the length of the list (depending on the given tag and the content of the given list), but it can never increase it. The `test_tag_filter/2` function is intended to describe this property of the `tag_filter/2` function. The `filter_list/0` function calls the `tag_filter/2` function to select the elements with key `integer` from the list `[{integer, 1}, {atom, a}]`.

Assume, that the programmer change the first element of the list in the function call `tag_filter([{integer, 1}, {atom, a}], integer)`). The impact of this modification flows into the `H2` variable, in the same way as in the first part of

```
tag_filter([], _Tag) ->
    [];
tag_filter([H2|T2], Tag) ->
    case H2 of
        {Tag, _Elem} -> [H2 | tag_filter(T2, Tag)];
        _    -> tag_filter(T2, Tag)
    end.

filter_list() ->
    tag_filter([{integer, 1}, {atom, a}], integer).

test_tag_filter(L2, Tag) ->
    len(L2) >= len(tag_filter(L2, Tag)).

len([]) ->
    0;
len([_H | T]) ->
    1 + len(T).
```

Fig. 2. The definition of `tag_filter/2`

our example where the elements of the `[1,2]` list flows into the `H1` variable, but after this point the difference between the `tag_add/2` and `tag_filter/2` functions shows up. The return value of the `tag_filter/2` function depends on the elements passed as its arguments, because the case-expression depends on the result of the `H2` expression, thus the return value of the `tag_filter/2`function depends on the elements of its parameter list (`H2` gets its value from that list). The expression `len(tag_filter(L2, Tag))` comprehends a function call of `tag_filter/2`, which return value depends on the elements of its parameter list, thus the result of the entire expression depends transitively on the elements of the argument list. Therefore, any change on the elements of the input list may have an impact on the `test_tag_filter/2` function.

To detect these dependencies in an Erlang program, we have to define a dependency graph for Erlang. It should contain the data flow edges and behaviour dependency edges, too. The rules when the value or the behaviour of an expression has an impact on an other expression (for example, the case expression in the mentioned example depends on the expression `H2`) are defined in the following sections.

3 A partial model for Erlang programs

In Section 4 we use the Erlang syntax shown in Figure 3. This syntax is a subset of the Erlang syntax presented in [4]. The symbol P denote the patterns can be used in Erlang, E represents guard expressions and expressions that can be defined in the language, and F denotes the named functions.

The presented syntax contains some simplification:

V ::= variables (including _, the underscore pattern)
A ::= atoms
I ::= integers
K ::= $A \mid I \mid$ other constants (e.g. strings, floats)
P ::= $K \mid V \mid \{P,\ldots,P\} \mid$ `[`$P,\ldots,P$`|`P`]`
E ::= $K \mid V \mid \{E,\ldots,E\} \mid$ `[`$E,\ldots,E$`|`E`]` $\mid$ `[`E`||`P`<-`E`]` $\mid P$ `=` $E \mid$
$E \circ E \mid$ `(`E`)` $\mid E$`(`$E,\ldots,E$`)` $\mid$
`case` E `of`
P when E `->` $E,\ldots,E$`;`
$\vdots$
P when E `->` $E,\ldots,E$
`end`
F ::= A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$`;`
$\vdots$
A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$.

Fig. 3. The used Erlang syntax subset

- Guard expressions are represented as expressions with some restrictions. Guard expressions can contain "guard" built-in function calls or type tests. The infix guard expressions are arithmetic or boolean expressions, or term comparisons. Guards can contain only bound variables.
- It does not contain those expression types which can be handled in the same way as one from the presented expressions. For example, the if and try construct can be handled similar as case expressions.
- Those language constructs which are not used to build the data dependency graph also left out from the model. For example, the attributes of an Erlang module do not hold relevant information in the meaning of data dependency.

4 Behaviour dependency graph

The most natural way to represent the impact of a change is a graph. To propagate dependency information we build a behaviour dependency graph (BDG).

4.1 The representation of the Erlang programs

To build the Erlang dependency graph we use the semantic program graph of RefactorErl. RefactorErl constructs the syntax tree representation of the source code and extends it with static semantic and lexical information. In RefactorErl each expression and pattern node is identified uniquely, we use these nodes as a base of the dependency graph, and the new edges represent the dependency information among them. While constructing the dependency graph we traverse the semantic graph, we take information from the graph, i.e. the structure of the syntax tree (expressions are attached to corresponding code parts), semantic

information (the binding structure of the variables, the function calls are linked to the definition of the function, etc). Just those syntactic nodes (mainly the expressions) appear in the dependency graph which are relevant in dependency propagation.

4.2 Dependency information

All the dependency information is represented in the behaviour dependency graph (BDG). The nodes of the graph are the expressions and patterns from the Erlang source code, the edges of the graph are representing dependency information. There are different kinds of dependency information, that is represented with labeled edges in the graph ($n_1 \overset{label}{\rightarrow} n_2$, where n_1 and n_2 are nodes of the graph). The different kinds of dependency edges are the followings:

Definition 1.

- **Data flow edges** – represent data flow between two nodes. There are different kinds of data flow information [7]:
  - Flow edges – $n_1 \overset{f}{\rightarrow} n_2$, represents that the result of n_2 can be a copy of the result of n_1. They value exactly the same, and changing the value of n_1 results the same change in the value of n_2.
  - Constructor edges – $n_1 \overset{c_i}{\rightarrow} n_2$, represents that the result of n_2 can be a compound value that contains n_1 as the ith element
  - Selector edges – $n_1 \overset{s_i}{\rightarrow} n_2$, represents that the result of n_2 can be the ith element of the n_1 compound data
- **Data dependency edges** – $n_1 \overset{d}{\rightarrow} n_2$, represents that the result of n_2 can directly depend on the result of n_1. Any change in n_1 may result a data or a behaviour change in n_2.
- **Behaviour dependency edges** – $n_1 \overset{b}{\rightarrow} n_2$, represents that the behaviour of n_2 can directly depend on the result of n_1. Any change in n_1 may result a behaviour change in n_2.

Note, that in case of constructing a list e is used as an element label, because we can not usually track their indexes([7]).

The change of a data has an impact on the behaviour of those expressions which depend on that data, thus each data dependency edge also represents behaviour dependency:

$$\frac{n_1 \overset{d}{\rightarrow} n_2}{n_1 \overset{b}{\rightarrow} n_2} \qquad \text{(d-b-rule)}$$

Similar, most of the flow edges propagate the change of the data, thus propagate dependency. Therefore there are nodes in the graph which are linked with multiply edges.

Examples The following example demonstrate the differences among the edge types.

```
e:
  case X of
     {ok, Result} -> Result + 2;
     _ when is_list(X) -> X
  end
```

There are different kinds of flow edges in case of this case expression: e. The result of the variable `X` simply flows to the tuple pattern: `X` $\xrightarrow{f}$ `{ok, Result}`, then while this pattern is a selector, it selects the elements from the tuple: `{ok, Result}` $\xrightarrow{s_1}$ `ok`, `{ok, Result}` $\xrightarrow{s_2}$ `Result`. The result if this case expression is the result of the last expression in its branches, so the result of the last expressions flow into the case expression: `Result+2` $\xrightarrow{f}$ e and `X` $\xrightarrow{f}$ e.

The result of the infix expression `Result+2` depends on the value of its subexpressions: `Result` $\xrightarrow{d}$ `Result+2` and `2` $\xrightarrow{d}$ `Result+2`.

This simple example also represent behaviour dependency information. The behaviour of the case expression is depend on the behaviour of its subexpressions. If the infix expression can not be evaluated then e also can not be evaluated: `Result+2` $\xrightarrow{b}$ e.

4.3 Dependency rules

As it is mentioned before, the dependency graph can be constructed based on the syntax tree and semantic information. The construction rules are summarized in Figures 4 and 5, and the major rules are described in the followings. The notation on the figures are: e is an expression (E), g is a guard expression (E), p is a pattern (P) and f is a function (F).

Variable. The only dependency among the variable bindings and the variable occurrences is the data flow (Figure 4: Variable). It does not hold data dependency, or behaviour dependency information.

Match expression Figure 4: Match exp. shows that the match expression contains a various number of dependency. The value of the expression e simply copied to the pattern p and to the expression e_0, that represented by flow edges. Each expression depends on the behaviour of its subexpressions, thus the match expression also represents behaviour dependency. The expression $p = e$ binds the value of e to p. In case if the variable p is already bound, then the match expression fails if the value of the variable p and the value of e do not match, so the result of the match expression e_0 may depend on the value of e, thus the match expression contains data dependency.

	Expressions	Graph edges
(Variable)	p is a binding n is a usage of the same variable	$p \xrightarrow{f} n$
(Match exp.)	e_0: $p = e$	$e \xrightarrow{f} e_0$, $e \xrightarrow{d} e_0$, $e \xrightarrow{b} e_0$ $e \xrightarrow{f} p$
(Pattern)	p_0: $p_1 = p_2$	$p_0 \xrightarrow{f} p_1$ $p_0 \xrightarrow{f} p_2$
(Infix exp.)	e_0: $e_1 \circ e_2$	$e_1 \xrightarrow{d} e_0$, $e_1 \xrightarrow{b} e_0$ $e_2 \xrightarrow{d} e_0$, $e_2 \xrightarrow{b} e_0$,
(Parenthesis)	e_0: (e)	$e \xrightarrow{f} e_0$, $e \xrightarrow{b} e_0$
(Tuple exp.)	e_0: $\{e_1, \ldots, e_n\}$	$e_1 \xrightarrow{c_1} e_0, \ldots, e_n \xrightarrow{c_n} e_0$ $e_1 \xrightarrow{b} e_0, \ldots, e_n \xrightarrow{b} e_0$
(Tuple pat.)	p_0: $\{p_1, \ldots, p_n\}$	$p_0 \xrightarrow{s_1} p_1, \ldots, p_0 \xrightarrow{s_n} p_n$
(List exp.)	e_0: $[e_1, \ldots, e_n \vert e_{n+1}]$	$e_1 \xrightarrow{c_\varsigma} e_0, \ldots, e_n \xrightarrow{c_\varsigma} e_0$, $e_{n+1} \xrightarrow{f} e_0$ $e_1 \xrightarrow{b} e_0, \ldots, e_n \xrightarrow{b} e_0$, $e_{n+1} \xrightarrow{b} e_0$
(List gen.)	e_0: $[e_1 \vert\vert p \leftarrow e_2]$	$e_1 \xrightarrow{c_\varsigma} e_0$, $e_2 \xrightarrow{s_\varsigma} p$ $e_1 \xrightarrow{b} e_0$, $e_2 \xrightarrow{b} e_0$
(List pat.)	p_0: $[p_1, \ldots, p_n \vert p_{n+1}]$	$p_0 \xrightarrow{s_\varsigma} p_1, \ldots, p_0 \xrightarrow{s_\varsigma} p_n$ $p_0 \xrightarrow{f} p_{n+1}$
(BIF 1)	e_0: $\mathsf{hd}(e_1)$	$e_1 \xrightarrow{s_\varsigma} e_0$ $e_1 \xrightarrow{b} e_0$
(BIF 2)	e_0: $\mathsf{tl}(e_1)$	$e_1 \xrightarrow{f} e_0$ $e_1 \xrightarrow{b} e_0$
(BIF 3)	I is constant, e_0: $\mathsf{element}(I, e_1)$	$e_1 \xrightarrow{s_I} e_0$ $e_1 \xrightarrow{b} e_0$

Fig. 4. Static behaviour dependency graph generation rules

Infix expressions The infix expression does not propagate data flow information, rather propagates data dependency information (Figure 4: Infix exp.). The result of an infix expression depends on the result of its subexpressions. If one of the subexpressions can not be evaluated, the infix expression can not be evaluated, so it also propagate behaviour dependency information.

Compound data structures. Beside data flow (flow, constructor and selector edges) information compound data structures (tuples, lists) also hold behaviour dependency information (Figures 4: Tuple exp., List exp. and List gen.). The behaviour of a compound data structure depends on the behavior of its elements, i.e. the expression depends on the behaviour of its subexpression.

	Expressions	Graph edges
(Case exp.)	e_0: case e of p_1 when $g_1 \rightarrow e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \rightarrow e_1^n, \ldots, e_{l_n}^n$ end	$e \xrightarrow{f} p_1, \ldots, e \xrightarrow{f} p_n$ $e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_n}^n \xrightarrow{f} e_0$ $e \xrightarrow{d} e_0$, $e \xrightarrow{b} e_0$ $e_1^1 \xrightarrow{b} e_0, \ldots, e_{l_1}^1 \xrightarrow{b} e_0$ $\vdots$ $e_1^n \xrightarrow{b} e_0, \ldots, e_{l_n}^n \xrightarrow{b} e_0$ $g_1 \xrightarrow{b} e_0, \ldots, g_n \xrightarrow{b} e_0$
(Fun. call 1)	e_0: f$(e_1, \ldots, e_n)$ f/n: f$(p_1^1, \ldots, p_n^1)$ when g_1 $\rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ f$(p_1^m, \ldots, p_n^m)$ when g_m $\rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_1 \xrightarrow{b} e_0, \ldots, e_n \xrightarrow{b} e_0$ $e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$ $e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$ $\vdots$ $e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$ $e_1^1 \xrightarrow{b} e_0, \ldots, e_{l_1}^1 \xrightarrow{b} e_0$ $\vdots$ $e_1^m \xrightarrow{b} e_0, \ldots, e_{l_m}^m \xrightarrow{b} e_0$ $g_1 \xrightarrow{b} e_0, \ldots, g_m \xrightarrow{b} e_0$
(Fun. call 2)	e_0: $e(e_1, \ldots, e_n)$ e is not constant, or e/n undefined	$e_1 \xrightarrow{d} e_0, \ldots, e_n \xrightarrow{d} e_0$, $e \xrightarrow{d} e_0$ $e_1 \xrightarrow{b} e_0, \ldots, e_n \xrightarrow{b} e_0$, $e \xrightarrow{b} e_0$

Fig. 5. Static behaviour dependency graph generation rules (cont.)

Conditional expressions. The behaviour of a conditional expression, like each complex expression, depends on its subexpressions. Thus each subexpression is linked to the expression with a behaviour dependency edge (Figure 4: Case exp.). For example, the case-expression depends on the behaviour of the expressions of its clauses, because an exception in these expressions propagate an exception into the case-expression.

Function calls. A function call similar to complex expressions, depends on its arguments, but it also depends on the body of the referred function (Figure 5: Fun. call 1). An exception from the body of the function has an impact on the function call expression, too. The result of an actual parameter flows into the corresponding formal parameter of the function, and the return value of the function (the result of the last expression of the function clause) flows back into the function call expression.

$e(e_1, \ldots, e_n)$ is an Erlang function call. In case when e is not a constant, we can not create dependency edges between the application and the function definition. The same situation occurs when the function is not defined in our graph – it is not added to the database(Figure 5: Fun. call 2). We handle that

case as a worst case scenario, and we generate data dependency edges among the function call and its subexpressions, because we do not know anything about the body of the function and the way how it uses and transforms the value of its parameters.

Built in functions (BIF). There are some built in function in Erlang which operate similar to the data selectors (Figure 4: BIF). For example the `hd/1` function selects the first elements of the list, or the `element/2` function selects the `I`-th element of a tuple. In these cases we add selector edges to the graph.

When the first parameter of the `element/2` function (`I`) is not a constant, the Function call 2 rule is applied.

4.4 Deriving dependency information

To determine the impact of a modification we need indirect/deeper dependency knowledge, thus we should calculate the transitive closure of the graph and traverse that graph. Each edge in the graph represent a dependency in the program, therefore when we want to determine the impact of a change, we have to traverse the graph using the corresponding defined edges.

A dependency relation between two graph nodes ($n_1 \rightsquigarrow n_2$) means the behaviour of n_2 depends on the result/behaviour of n_1, so the change of the value of n_1 may have an impact on n_2. This relation can be computed using the data flow, data dependency and the behaviour dependency edges.

The informal definition of the dependency relation $n_1 \rightsquigarrow n_2$ is that n_2 is an expression in the graph which could be affected by changing the value of n_1. Those nodes from the graph which could be a copy of n_1 are affected by changing the value of n_1, so modifying n_1 could have an impact on them. Therefore the data flow propagate the changes (data-rule).

Consider the following expression: `1+2`. Changing the expression `1` to `atom` results that the expression `1+2` could not be evaluated and that results a runtime error. Then each expression which behaviour depend on the value of `1+2` also could not be evaluated. Therefore when there is data dependency connection between two nodes ($n_1 \xrightarrow{d} n_2$), changing the data in n_1 could have an impact on the behaviour of n_2, and those node which behaviour may depend from n_2, also could alter behaviour from the same data change (b-dep-rule). Data flow and the behaviour dependency edges ($\xrightarrow{b}$) also propagates behaviour dependency among expressions (d-rule, b-rule).

In the followings we formalize the mentioned behaviour dependency relation.

Definition 2. The data flow relation $\overset{d}{\rightsquigarrow}$ is defined as the minimal that satisfies the following rules [7]:

$$n \overset{d}{\rightsquigarrow} n \qquad \text{(reflexive)}$$

$$\frac{n_1 \xrightarrow{f} n_2}{n_1 \overset{d}{\rightsquigarrow} n_2} \qquad \text{(f-rule)}$$

$$\frac{n_1 \xrightarrow{c_i} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3,\ n_3 \xrightarrow{s_i} n_4}{n_1 \overset{d}{\rightsquigarrow} n_4} \quad \text{(c-s-rule)}$$

$$\frac{n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3}{n_1 \overset{d}{\rightsquigarrow} n_3} \quad \text{(transitive)}$$

Definition 3. The behaviour dependency relation $\overset{b}{\rightsquigarrow}$ is defined as the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{d}{\rightsquigarrow} n_2}{n_1 \overset{b}{\rightsquigarrow} n_2} \quad \text{(d-rule)}$$

$$\frac{n_1 \overset{b}{\rightsquigarrow} n_2,\ n_2 \xrightarrow{b} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_4}{n_1 \overset{b}{\rightsquigarrow} n_4} \quad \text{(b-rule)}$$

Definition 4. The dependency relation $\rightsquigarrow$ is defined as the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{d}{\rightsquigarrow} n_2}{n_1 \rightsquigarrow n_2} \quad \text{(data-rule)}$$

$$\frac{n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \xrightarrow{d} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_4}{n_1 \rightsquigarrow n_4} \quad \text{(b-dep-rule)}$$

4.5 Lemmas

In this section some lemmas about the properties of relations $\overset{b}{\rightsquigarrow}$ and $\rightsquigarrow$ are introduced. The detailed proofs of the lemmas are presented in appendix A.

Lemma 1 ($\overset{b}{\rightsquigarrow}$ reflexive).
$n \overset{b}{\rightsquigarrow} n$

Proof. Applying the rules (reflexive) and (d-rule).

Lemma 2 ($\overset{b}{\rightsquigarrow}$ transitive).
$n_1 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

Proof. Applying structural induction on $n_1 \overset{b}{\rightsquigarrow} n_2$ and then structural induction on $n_2 \overset{b}{\rightsquigarrow} n_3$.

Lemma 3 ($\rightsquigarrow$ reflexive).
$n \rightsquigarrow n$

Proof. Applying the rules (reflexive) and (data-rule).

Lemma 4 ($\rightsquigarrow$ transitive).
$n_1 \rightsquigarrow n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$

Proof. Applying case distinction on $n_1 \rightsquigarrow n_2$ and then case distinction on $n_2 \rightsquigarrow n_3$.

Lemma 5 (generalized b-dep-rule).
$n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \rightsquigarrow n_4 \Rightarrow n_1 \rightsquigarrow n_4$

Proof. Applying case distinction on $n_3 \rightsquigarrow n_4$.

Lemma 6 ($\overset{b}{\rightsquigarrow}$ is not symmetrical and is not anti-symmetrical).

Proof. See the counterexamples in appendix A.

4.6 Example

Figures 6 and 7 shows the relevant part of the dependency graphs for the motivation examples.

The main difference between them is the `H2` $\overset{d}{\rightarrow}$ `case H2 of ... end` edge, which describes that the result and the behaviour of the case-expression depends on `H2`.

On the left hand side figure, the value `1` simply flows (as a meaning of data flow) into the variable `_H` (in `len/1`). The `len/1` function does not use the value of `_H`, so its return value does not depend on it. On the other figure the value of the tuple `{integer, 1}` flows into the variable `H2` (in `tag_filter/2`). The case-expression depends on the value of `H2`, the value of the case-expression copied to the result of `tag(L1, Tag)` (in `len/1`), so the change of the tuple may have an impact on the `len/1` function.

In summary, if we modify the integer `1` we must not the `test_tag_add/2` test case, but if we modify the tuple `{integer, 1}`, we should run test case `test_tag_filter/2`.

5 Related work

A methodology for regression test selection in object oriented designs have been already presented in [1]. That methodology represents the designs using the Unified Modeling Language, and gives a formal mapping between design changes and a classification of regression test cases (reusable, retestable, obsolete).

Our model tries to find affected test cases using a graph traversal on the BDG. The BDG adds behaviour edges according to the semantics of the Erlang language to a 0-th order Data Flow Graph (0DFG). Most of the data flow and the behaviour flow edges are specific for Erlang (some of them are discussed in Section 4.3) which do not appears in other languages. Specially, the behaviour edges manly refers to exceptions (that can arise during the evaluation after a data change) in our model. In the analysis of other languages this kind of edges usually do not appear with a data flow graph, rather just the exception

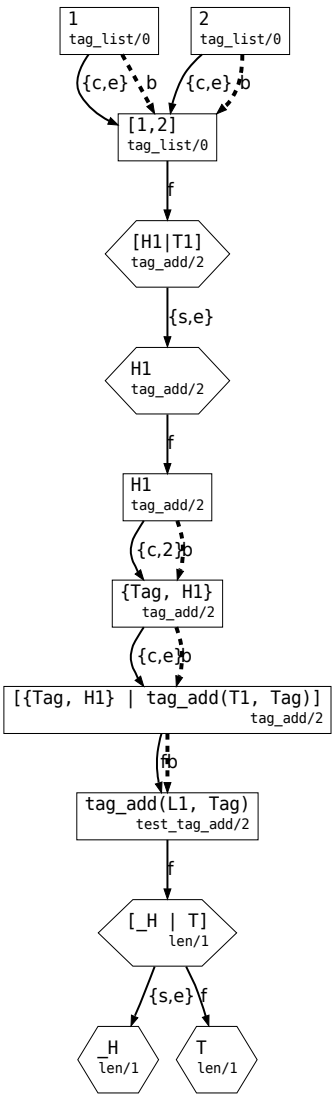


Fig. 6. Dependencies in tag_add

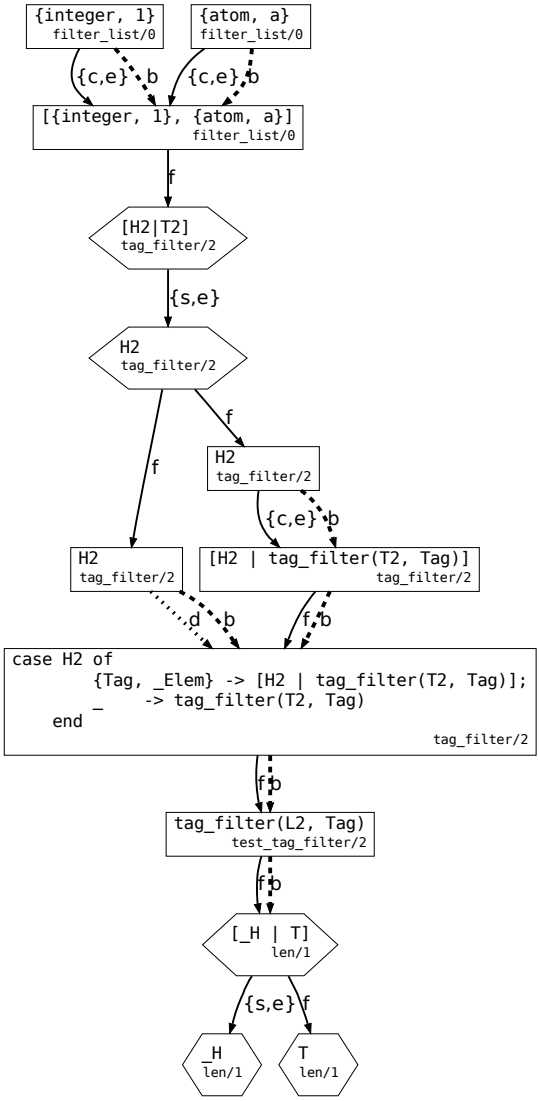


Fig. 7. Dependencies in tag_filter

handling constructs are handled in a control flow graph [9]. Our model could be applicable to other strict functional languages to detect the spread of a data change, however when applying it to a lazy language further analysis could be useful.

Estimating the impact of a change in functional programming languages is not really widespread yet, however control flow analysis have been already studied by Shivers [8], but this work applied for optimizing compilers. Data flow analysis already defined for Erlang [7] and successfully applied to module interface upgrade. For Erlang the Control flow analysis successfully applied for improving testing [10].

6 Conclusions and future work

In this paper we present a dependency graph to calculate the impact of some modification in an Erlang source code. The base idea behind this is to support the programmers to reduce the number of test cases which should be performed after a refactoring transformations. Therefore we have to propagate the change made by the transformation in a behaviour dependency graph. This graph contains the relevant expression nodes from the syntax tree and data flow, data dependency and behaviour dependency edges. The result of the dependency graph has been illustrated in the motivating example.

The size of the presented graph is linear to the size of the syntax tree. If there are n expression nodes in the syntax tree, the size of the graph is $O(n)$, the size of its transitive closure is maximum $O(n^2)$.

This paper shows a structural algorithm to construct the BDG and a relation to calculate the dependency in that graph. The DFG is already implemented in the RefactorErl system, thus we should add the behaviour edges to that graph and implement the dependency relation. Then we could examine the efficiency of our model.

We can improve the presented solution in different ways. The presented model does not make a distinction among the different calls of a function, and the return value of the function is linked to each function call. The problem with that approach is that when we call the function, we reach the result of other function calls, too. To solve this problem we should store context information about the source, i.e. where is the function called. Therefore, more accurate graph could be generated using 1CFA-s [8] or nCFA-s, as the behaviour dependency edges could be generalized according to the order of the analysis, so labeled edges could be used in our BDG to represent the context information. We note, that our model based on a 0DFG and adds behaviour edges to that graph, and it results less test case subset in our example, but using 1DFG-s could also result less test case subset than using 0DFG-s.

When we generate a BDG, it could grow fast, and could be unnecessary huge. Therefore, we should trim the irrelevant parts from the graph. A possible solution should be to combine the control flow with the result of a call graph. First we can create a call graph part from the change, and then we can should create the data flow and the control flow using the affected functions from the call graph. We can build the whole dependency graph and then trim it (for example with slicing), or just build the smaller graph. Thus we can calculate other analysis and iterative algorithms on smaller graphs.

References

1. L. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *18th IEEE International Conference on Software Maintenance (ICSM'02)*, 2002.
2. Ericsson AB. *Erlang Reference Manual.* Latest version available online at http://www.erlang.org/doc/reference_manual/part_frame.html.
3. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
4. L.-A. Fredlund. *A Framework for Reasoning about ERLANG code.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.
5. Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, M. Tóth, I. Bozó, and R. Király. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babe-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, Jul 2009.
6. Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, and R. Király. Building a refactoring tool for erlang. In *Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008*, Paphos, Cyprus, Jul 2008.
7. L. Lövei. Automated module interface upgrade. In *Erlang '09: Proceedings of the 8th ACM SIGPLAN workshop on Erlang*, pages 11–22, New York, NY, USA, 2009. ACM.
8. O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.
9. S. Sinha and M. J. Harrold. Control-flow analysis of programs with exception-handling constructs. Technical report, 1998.
10. M. Widera. Flow graphs for testing sequential erlang programs. In *Proceedings of the ACM SIGPLAN 2004 Erlang Workshop*, pages 48–53, 2004.

A The detailed proofs of lemmas.

Lemma 1 ($\overset{b}{\rightsquigarrow}$ reflexive).
$n \overset{b}{\rightsquigarrow} n$

Proof.

$$\frac{n \overset{d}{\rightsquigarrow} n \text{ (reflexive)}}{\models\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=} \text{ (d-rule)}$$
$$n \overset{b}{\rightsquigarrow} n$$

□

Lemma 2 ($\overset{b}{\rightsquigarrow}$ transitive).
$n_1 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

Proof. structural induction on $n_1 \overset{b}{\rightsquigarrow} n_2$

a) (base step) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$
structural induction on $n_2 \overset{b}{\rightsquigarrow} n_3$
i) (base step) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

$$\dfrac{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3}{\models\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=} \text{ (transitive)}}{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_3}{\models\!=\!=\!=\!=\!=\!=\!=\!=\!=} \text{ (d-rule)}}$$
$$n_1 \overset{b}{\rightsquigarrow} n_3$$

ii) (induction step) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

$$\dfrac{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3}{\models\!=} \text{ (induction hypothesis)}}{\dfrac{n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3}{\models\!=} \text{ (b-rule)}}$$
$$n_1 \overset{b}{\rightsquigarrow} n_3$$

b) (induction step) $n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

$$\dfrac{\dfrac{n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3}{\models\!=} \text{ (induction hypothesis)}}{\dfrac{n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3}{\models\!=} \text{ (b-rule)}}$$
$$n_1 \overset{b}{\rightsquigarrow} n_3$$

□

Lemma 3 ($\rightsquigarrow$ reflexive).
$n \rightsquigarrow n$

Proof.

$$\dfrac{n \overset{d}{\rightsquigarrow} n \text{ (reflexive)}}{\models\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=} \text{ (data-rule)}$$
$$n \rightsquigarrow n$$

□

Lemma 4 ($\rightsquigarrow$ transitive).
$n_1 \rightsquigarrow n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$

Proof.
case distinction on $n_1 \rightsquigarrow n_2$

a) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$
case distinction on $n_2 \rightsquigarrow n_3$

i) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3 \Rightarrow n_1 \rightsquigarrow n_3$

$$\dfrac{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3}{\models\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=} \text{ (transitive)}}{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_3}{\models\!=\!=\!=\!=\!=\!=\!=\!=\!=} \text{ (data-rule)}}$$
$$n_1 \rightsquigarrow n_3$$

ii) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \rightsquigarrow n_3$

$$\dfrac{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3}{\models\!=} \text{ (transitive)}}{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3}{\models\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=} \text{ (b-dep-rule)}}$$
$$n_1 \rightsquigarrow n_3$$

b) $n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$
case distinction on $n_2 \rightsquigarrow n_3$

i) $n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3 \Rightarrow n_1 \rightsquigarrow n_3$

$$\dfrac{\dfrac{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3}{\models\!=} \text{ (d-rule)}}{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3}{\models\!=} \ (\overset{b}{\rightsquigarrow} \text{ transitive})}}{\dfrac{n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3}{\models\!=} \text{ (b-dep-rule)}}$$
$$n_1 \rightsquigarrow n_3$$

ii) $n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \rightsquigarrow n_3$

$$\begin{array}{l} n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2, \\ n_2 \overset{d}{\rightsquigarrow} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\rightsquigarrow} n_3 \\ \models\!\!= \text{ (d-rule)} \\ n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2, \\ n_2 \overset{b}{\rightsquigarrow} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\rightsquigarrow} n_3 \\ \models\!\!= \text{ ($\overset{b}{\rightsquigarrow}$ transitive)} \\ n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\to} n_5, \\ n_5 \overset{b}{\rightsquigarrow} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\rightsquigarrow} n_3 \\ \models\!\!= \text{ (d-b-rule)} \\ n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\to} n_5, \\ n_5 \overset{b}{\rightsquigarrow} n_6,\ n_6 \overset{b}{\to} n_7,\ n_7 \overset{b}{\rightsquigarrow} n_3 \\ \models\!\!= \text{ (b-rule)} \\ n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3 \\ \models\!\!= \text{ (b-dep-rule)} \\ n_1 \rightsquigarrow n_3 \end{array}$$

□

Lemma 5 (generalized b-dep-rule).
$n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \rightsquigarrow n_4 \Rightarrow n_1 \rightsquigarrow n_4$

Proof.
case distinction on $n_3 \rightsquigarrow n_4$

a) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_4 \Rightarrow n_1 \rightsquigarrow n_4$

$$\begin{array}{l} n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_4 \\ \models\!\!= \text{ (b-rule)} \\ n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_4 \\ \models\!\!= \text{ (b-dep-rule)} \\ n_1 \rightsquigarrow n_4 \end{array}$$

b) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_5,\ n_5 \overset{d}{\to} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 \Rightarrow n_1 \rightsquigarrow n_4$

$$\begin{array}{l} n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_5,\ n_5 \overset{d}{\to} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 \\ \models\!\!= \text{ (d-b-rule)} \\ n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_5,\ n_5 \overset{b}{\to} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 \\ \models\!\!= \text{ (d-rule)} \\ n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_5,\ n_5 \overset{b}{\to} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 \\ \models\!\!= \text{ (b-rule)} \\ n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_4 \\ \models\!\!= \text{ (b-dep-rule)} \\ n_1 \rightsquigarrow n_4 \end{array}$$

□

Lemma 6 ($\overset{b}{\rightsquigarrow}$ is not symmetrical and is not anti-symmetrical).

Proof. Lets consider the following examples:
Example 1.

```
ten()-> 10.
add_ten(X) -> X + ten().
```

$\overset{b}{\rightsquigarrow}$ is not symmetrical if exist two expression in the graph n_1 and n_2 where $n_1 \overset{b}{\rightsquigarrow} n_2$ but *not* $n_2 \overset{b}{\rightsquigarrow} n_1$. If n_1 is the integer `10` from the body of `ten/0` and n_2 is the function call `ten()` in the body of `add_ten/1` then:

$$\frac{n_1 \overset{b}{\rightsquigarrow} n_1,\ n_1 \overset{b}{\rightarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_2}{\models\!=\ \text{(b-rule)}\quad n_1 \overset{b}{\rightsquigarrow} n_2}$$

Based on the behaviour dependency graph building rules (Figures 4. and 5.), only two edges start from n_2: a $\overset{b}{\rightarrow}$ and a $\overset{d}{\rightarrow}$ edge, both to the direction of the infix expression `X + ten()`. There is now direct edges or graph paths starting from the expression `X + ten()`, thus does not exist any path from n_2 to n_1, so *not* $n_2 \overset{b}{\rightsquigarrow} n_1$.
Example 2.

```
f(0) -> 0;
f(A) when A > 0 -> f(A) - 1.
```

$\overset{b}{\rightsquigarrow}$ is not anti-symmetrical if exist two expression in the graph n_1 and n_2 where $n_1 \overset{b}{\rightsquigarrow} n_2$ and $n_2 \overset{b}{\rightsquigarrow} n_1$. If n_1 is the infix expression `f(A)-10` and n_2 is the function call `f(A)` then both $n_1 \overset{b}{\rightsquigarrow} n_2$ and $n_2 \overset{b}{\rightsquigarrow} n_1$ are true:

$$\frac{n_1 \overset{f}{\rightarrow} n_2}{\models\!=\!=\!=\!=\!=\!=\!=\!=\ \text{(f-rule)}\quad n_1 \overset{d}{\rightsquigarrow} n_2}\ \models\!=\!=\!=\!=\!=\!=\!=\!=\ \text{(d-rule)}\quad n_1 \overset{b}{\rightsquigarrow} n_2$$

$$\frac{n_2 \overset{b}{\rightsquigarrow} n_2\ (\overset{b}{\rightsquigarrow} reflexive),\ n_2 \overset{b}{\rightarrow} n_1,\ n_1 \overset{b}{\rightsquigarrow} n_1\ (\overset{b}{\rightsquigarrow} reflexive)}{\models\!=\ \text{(b-rule)}\quad n_2 \overset{b}{\rightsquigarrow} n_1}$$

$n_1 \overset{f}{\rightarrow} n_2$ is based on the rule *Fun call 1.* and $n_2 \overset{b}{\rightarrow} n_1$ is based on the rule *Infix exp.*

□

C. függelék

Párhuzamosítható minták felismerése

A függelékben szereplő cikkek [TBK17, BFH+15, BFH+14, KTBH16] végleges változatának publikációs adatai:

- Melinda Tóth, István Bozó, and Tamás Kozsik. Pattern Candidate Discovery and Parallelization Techniques. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages*, p. 1-26, New York, NY, USA, 2017. ACM.

- István Bozó, Viktória Fördős, Dániel Horpácsi, Zoltán Horváth, Tamás Kozsik, Judit Kőszegi, and Melinda Tóth. Refactorings to enable parallelization. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming*, pages 104–121, Cham, 2015. Springer International Publishing.

- István Bozó, Viktoria Fordós, Zoltán Horváth, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. Discovering parallel pattern candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, New York, NY, USA, 2014. ACM.

- Tamás Kozsik, Melinda Tóth, István Bozó, and Zoltán Horváth. Static analysis for divide-and-conquer pattern discovery. *Computing and*

Informatics, 35:764–791, 01 2016.

Pattern Candidate Discovery and Parallelization Techniques

Melinda Tóth, István Bozó, Tamás Kozsik
ELTE, Eötvös Loránd University
Budapest, Hungary
{tothmelinda,bozoistvan,kto}@elte.hu

ABSTRACT

Parallel computations in a program can be expressed conveniently, at a high level of abstraction, using parallel patterns such as task farm, pipeline or divide-and-conquer. In order to transform a sequential program into a pattern-based parallel one, the software developer may want to apply refactoring transformations on it. This tutorial explains a methodology to perform tool supported parallelization of programs by presenting how to use a specific static source code analysis and transformation system for Erlang. A key element of the approach is pattern candidate discovery, a static analysis technique to identify code fragments that can be refactored into a parallel pattern.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Software development techniques**;

KEYWORDS

pattern based programming, Erlang, PaRTE, static analysis, refactoring

ACM Reference Format:
Melinda Tóth, István Bozó, Tamás Kozsik. 2017. Pattern Candidate Discovery and Parallelization Techniques. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages, August 30-September 1, 2017, Bristol, United Kingdom.* ACM, New York, NY, USA, 27 pages. https://doi.org/10.1145/3205368.3205369

1 INTRODUCTION

Refactoring [12, 41] is the process of restructuring, shaping or transforming a program in order to improve its quality, to change its non-functional properties, or to make it suitable to add a new feature. Software developers often carry out this activity by hand, which is a tiring and error-prone process. For this reason, the use of refactoring tools is considered beneficial. Refactoring tools for mainstream languages are available as part of the widespread integrated software development environments. They mostly focus on source code transformations related to the object-oriented paradigm: they allow the manipulation of classes, class members and the inheritance relationship. There are some refactoring tools also for functional programming languages, such as HaRe [29] for Haskell, or Wrangler [29] and RefactorErl [8] for Erlang.

Possible application areas of refactoring are the introduction of parallelism into a sequential program [16], and the transformation of a parallel program into one with a different structure of parallelism. This latter is extremely useful when the software developer is experimenting with the code, trying to identify the best, most efficient parallel structure for a given software on a given hardware.

Pattern based parallelism is great for this experimentation. The structure of parallel computations in a program can be defined conveniently, and at a high level of abstraction, using parallel patterns [13, 14]. Some well-known patterns are the task farm, the pipeline, the map-reduce and the divide-and-conquer patterns. An algorithmic skeleton library can provide implementations for the patterns, and therefore it allows the programmer to separate the *what* and the *how* in a parallel program: the computations to be carried out on the one hand, and the applied parallel structure on the other. In order to change this parallel structure, it is often enough to switch from one algorithmic skeleton to another one.

Being able to perform refactoring transformations with a tool, quickly and safely, is already a good thing. But how do we know *where* to apply a transformation, where to introduce a parallel pattern in some sequential code? This question is essential in the case of large, industrial-scale software applications. When parallelizing large code bodies, a tool offering guidance to its user on what refactoring decisions are to be made, on where it is the most fruitful to introduce parallelism, and on how to achieve the desired program structure, can provide invaluable help. Software developers may be unfamiliar with all the small details of the code they work on, and hence it would take lots of time for them to find the parts that are possible, and *worth*, to parallelize. This paper presents a software tool, the ParaPhrase Refactoring Tool for Erlang (*PaRTE*, pronounced as "party"), which has been designed with the above ideas in mind. Not only is it capable of performing refactoring transformations, but it also advises software developers of the promising opportunities for parallelization. This feature, *pattern discovery*, identifies pattern candidates, and in certain cases it is also able to tell

IFL 2017, August 30-September 1, 2017, Bristol, United Kingdom

ACM ISBN 978-1-4503-6343-3/17/08.
https://doi.org/10.1145/3205368.3205369

which candidate seems more profitable in terms of potential parallel speedup.

ParaPhrase is the (short) name of an EU FP7 project, which produced a modern, structured design and implementation approach for parallel programming, allowing developers exploit a variety of high-level parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically re-mapped to meet application needs and hardware availability [23]. The project provided paraphrasing infrastructure for two programming languages, C++ and Erlang, among which Erlang is the one supported by PaRTE.

The code examples in this paper are also written in Erlang. There are excellent books discussing this language, e.g. [2, 24]. In a nutshell, Erlang is a fairly simple, but powerful, dynamically typed, impure functional language with strict evaluation, compiling to, and running in, a virtual machine, the Erlang VM. One of its strengths is its set of scalable and easy-to-use concurrency features. Processes are cheap to create, are preemptively scheduled, and communicate with message passing. It is also straightforward and quite transparent to distribute those processes in a network of Erlang VMs. Efforts to further scale distributed Erlang systems have also been made recently [5].

Another strength of Erlang lies in its philosophy concerning fault tolerance. Its approach to program design is backed by Erlang/OTP, which is "simultaneously a framework, a set of libraries, and a methodology for structuring applications" [30]. According to this philosophy, application components must be organized into processes so that they become really loosely coupled, and in the case of a failure they can be easily stopped and re-started.

Although Erlang is not particularly well-suited for computation-intensive programs (and in practice it is rarely used for that purpose), it offers a nice language environment to study pattern-based parallelism. Moreover, its concurrency features allowed the efficient implementation of an algorithmic skeleton library, *Skel*, which provides access to a number of parallel patterns [43], and even supports heterogeneous (mixed CPU and GPU) computations [28].

The rest of this paper is structured as follows. Section 2 familiarizes you with Erlang (including the syntax for functions, lists, tuples and lambdas, processes and communication primitives etc.). Moreover, this section presents a motivational example for a parallel pattern candidate. Section 3 covers the fundamentals of pattern based parallel programming, showing some well-known parallel patterns and how they can be represented with the Skel library. Section 4 demonstrates the use of the ParaPhrase Refactoring Tool for Erlang, which can discover parallel pattern candidates, and help refactor them into applications of Skel-based algorithmic skeletons. Section 5 explains the inner workings of PaRTE, focusing on the static source code analyses which are at the heart of pattern candidate discovery. The tool and the methodology are evaluated on a number of real-world, open source code bases in Section 6. Section 7 discusses related work, and finally, Section 8 concludes the paper.

The main goal of this paper is to provide a self-contained, profound presentation of the methodology behind the ParaPhrase Refactoring Tool for Erlang: the underlying concepts of pattern based parallelism, the refactoring approach to introduce algorithmic skeletons, and – with the strongest emphasis – the pattern candidate discovery technique.

An important technical contribution of this paper is the detailed description of the analyses applied by PaRTE for pattern candidate discovery, which can be used to design similar tools for other programming languages as well. Discovery rules for some of the patterns can be found in earlier articles, i.e. [6, 7, 26] (these are briefly presented here for completeness), but rules to identify many other kinds of candidates are also shown in this paper.

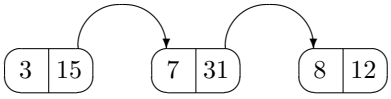
2 EXAMPLE: A PATTERN CANDIDATE

Some people use the terms *parallel* and *concurrent* interchangeably. Others prefer to separate these two concepts [35], saying that

- concurrency is when you define a system with a number of communicating, co-operating processes, typically exhibiting non-determinism;
- parallelism is when you speed up a computation by performing it on a number of processing elements – but otherwise you arrive at the same result as a sequential (and deterministic) computation.

We argue that the two concepts should be approached with different programming techniques. Concurrency might be too low-level to achieve parallelism, and may become a bottleneck in effective software development. Let us explain this by using a parallel programming example: we shall define a function implementing sparse matrix-vector multiplication.

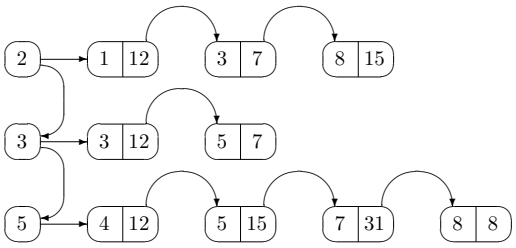
A sparse matrix is a matrix with lots of zero elements. One can efficiently represent such a matrix as a list of its non-zero elements and their position. Similarly, a sparse vector can be represented as a list of pairs, where each pair contains an index and a (non-zero) value. Consider the following list, which contains three pairs.



This structure can be written as an Erlang *term* using the *list* constructor `[]` and the *tuple* constructor `{}` as like this: `[{3,15}, {7,31}, {8,12}]`. Assuming 0-based indices, the above list can represent the following vector (of length 10).

0	0	0	15	0	0	0	31	12	0

Note that the list of pairs is sorted by the first (i.e. the index) component – this will be very helpful for implementing operations, in our case the multiplication. First of all, let us see how to multiply two sparse vectors r and c to compute



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 12 & 0 & 7 & 0 & 0 & 0 & 0 & 15 \\ 0 & 0 & 12 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 15 & 0 & 31 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 1: Sparse matrix representation

their dot-product, $r_i c_i$. Note that we need to sum up $r_i c_i$ for all i indices such that $r_i \neq 0$ and $c_i \neq 0$.

The `vxv/2` (that is *vector-times-vector*) function[1] takes two lists that represent the two sparse vectors, and computes the dot-product recursively, by traversing the two lists simultaneously. When both vectors contain a non-zero value at a given index position i, they are multiplied, and the product is added to the final result – as described by the first *clause* of the following definition.

```
vxv( [{I,R} | Rs], [{I,C} | Cs] ) ->
  R*C + vxv(Rs,Cs);
vxv( [{I,R} | Rs], [{J,C} | Cs] ) ->
  if
    I<J  -> vxv( Rs, [{J,C}|Cs]);
    true -> vxv( [{I,R} | Rs], Cs )
  end;
vxv( _, _ ) ->     % when either (or both)
  0.               % of the lists is empty
```

The second clause of the function definition tells us what to do when the head of the two input lists corresponds to non-empty vector elements `R` and `C` at different index positions (`I` and `J`, respectively). Now we make use of the fact that the lists are sorted by the index component. If `I < J`, the non-zero elements stored in the right list are all at an index position larger than `I`, and hence `R` is not needed for the dot-product: this corresponds to $r_i c_i$ with $r_i \neq 0$ and $c_i = 0$ in the sum. Therefore, we discard the head of the left list, and proceed recursively. Similarly, when `I > J`, we discard the head of the right list, and proceed recursively.

The last clause of `vxv/2` is reached when either of the lists is empty. In this case, the corresponding sparse vector does not contain any more non-zero elements, and hence the dot-product is 0.

Now we are ready to implement the matrix-vector multiplication, the `mxv/2` function. A sparse matrix is represented as a list of those rows that contain at least one non-zero element. Each such row in the matrix is represented as the pair of the row index, and a (non-empty) list representing a sparse vector (Figure 1).

The product of a matrix and a (column) vector is a (row) vector, the elements of which are the dot-product of a row vector in the matrix and the given column vector. The row vector is stored in variable `DotProducts` (in Erlang the name of a variable must start with a capital letter, while the name of a function must start with a lowercase letter). The last step of the computation is to drop zero values from `DotProducts` using the `filter/2` function from the standard library *module* `lists`.

```
mxv( Rows, Col ) ->
  DotProducts =
    [ {I,vxv(Row,Col)} || {I,Row} <- Rows ],
  lists:filter( fun({_I,V}) -> V /= 0 end,
                DotProducts).
```

The `lists:filter/2` function is a higher-order function: its first argument is a predicate, in this case an *explicit fun-expression* (often called as lambda-expression in other languages). This specific predicate takes a pair, discards its first element (we should start the name of unused parameters/variables with an underscore), and returns true *iff* the second element is non-zero.

2.1 Compute faster with processes!

It is very easy to speed up the computation of `mxv/2` by spawning processes, and let them calculate the dot-products in parallel. One can easily rewrite `mxv/2` to spawn a process for each represented row in the matrix, and instruct those processes to calculate the dot-products using `vxv_proc/3` (a wrapper around `vxv/2`). In the rewritten `mxv/2` function, we bind the list of row indices (`I`) and the corresponding process identifiers (`Pid`) to the variable `IndicesAndPids`. The dot-products are communicated back from the processes and collected by the **`receive`** expression in the spawning order of the processes, and hence in the order of row indices.

```
mxv( Rows, Col ) ->
  IndicesAndPids =
    [ {I, spawn(?MODULE, vxv_proc,
                [self(), Row, Col]
      || {I,Row} <- Rows ],
  DotProducts =
    [ receive
        {Pid, Res} -> {I, Res}
      end
      || {I, Pid} <- IndicesAndPids ],
  lists:filter( fun({_,V}) -> V /= 0 end,
                DotProducts ).
```

[1] Erlang supports overloading for functions, i.e. we can have functions with the same name, but with different arity. To distinguish between overloaded functions, the *name/arity* notation is used, like in `vxv/2`.

Dot-products are computed in parallel by processes that execute `vxv_proc/3`. When a new process is spawned, we pass the process identifier of the "master" process, `self()`, together with the row vector and the column vector to this function. This `vxv_proc/3` simply calls `vxv/2`, and sends back the computed dot-product (together with the process id of the "slave" process, `self()`) to the parent process.

```
vxv_proc(ParentPid, Row, Col) ->
  ParentPid ! {self(), vxv(Row, Col)}.
```

The concurrency features of Erlang allowed us to parallelize the sparse matrix-vector multiplication by introducing processes, spawning them, and implementing communication among them based on the process identifiers. One may consider this a relatively small overhead to arrive at a parallel computation. However, there are a number of small stupid mistakes one can make. For example, consider the two calls to `self/0`. This impure function returns the *pid* of the process that evaluates the call. The two calls to `self/0` will be evaluated by different processes here: the one in `mxv/2` is evaluated in the master process, while the other one in the spawned processes. The lack of referential transparency can be rather confusing in this case.

Another opportunity for shooting yourself on the foot is the following. One could easily come to the idea that the definition of `mxv/2` can be refactored, and the two variables can be eliminated. These variables are used only once, so why not replace them with their definition?

```
mxv( Rows, Col ) ->
  lists:filter(
    fun({_,V}) ->
      V /= 0 end,
      [ receive
          {Pid, Res} -> {I, Res}
        end
        || {I, Row} <- Rows,
           Pid <- [spawn(?MODULE, vxv_proc,
                         [self(),Row,Col])]
      ]
    ).
```

Of course, it is a matter of taste whether we consider this new definition simpler or more obscure – but this refactoring might seem reasonable at first glance. However, it turns out that there is no parallel computation in this solution any more. It is true that the list generator in the second argument of `lists:filter/2` starts a new process for each row, and receives the dot-product of this row and `Col` from this process. But note that the next process is not spawned until the result from the previous one is received. There is an unnecessary synchronization in this definition that prevents parallel execution. The refactoring we applied was valid in the sense that this new definition produces exactly the same result as the previous one, hence the definitions can be regarded as semantically equivalent. But obviously, losing parallelism was not intended!

2.2 Simply parallel

Now we can see that implementing parallel computations using the concurrency features of a language can be error-prone, and sometimes really cumbersome. In the following we will see how to write a parallel program in a way that avoids this low-level trickery with concurrency, synchronization, communication, process management and so on. This new technique will allow us to describe a parallel computation at a high level of abstraction, in terms of common solution recipes, *parallel patterns*. Intuitively, the structure of the parallel computation for our sparse matrix-vector multiplication, `Computation`, can be given as a task farm with `length(Rows)` tasks, each task performing a "sequential" component `Function`, and forcing an ordered collection of results. The computational structure is described using *algorithmic skeletons* (`ord`, `farm`, `seq`). The computation is carried out by the `skel:do/2` function, the main entry point to the Skel library.

```
mxv( Rows, Col ) ->
  Function =
    fun ({I,Row}) -> {I,vxv(Row,Col)} end,
  Computation =
    [{ord,[{farm,[{seq,Function}],
            length(Rows)}]}],
  DotProducts = skel:do( Computation, Rows),
  lists:filter( fun({_I,V}) -> V /= 0 end,
                DotProducts).
```

In the forthcoming sections we cover some details of parallel patterns, and discover how Skel, an algorithmic skeleton library for Erlang allows us to express the structure of parallel computations.

3 PARALLEL PATTERNS

Some of the most well-known patterns for parallel computations are the task farm, the pipeline and the divide-and-conquer patterns. This section will focus on these, and show how to express them with the Skel library. Then, in later sections, the technique to discover candidates for these patterns will be presented.

However, there are many other interesting patterns, such as the *map*, the *reduce*, the *feedback*, the *orbit* and the *stencil* patterns, which, by the way, can also be expressed with Skel. The interested reader is referred to [13, 14, 43].

3.1 Task farm

The task farm pattern models embarrassingly stream-parallel computations, where a worker function is iterated over the elements of the input. This sort of parallel execution is possible whenever the computation can be processed on each item of the input independently of the computations on other items of the input. In the previous section, we could see the sparse matrix-vector multiplication function, which exemplifies this behaviour: the dot-product computation is applied on each row of the sparse matrix, independently of the other rows.

```
mxv( Rows, Col ) ->
  DotProducts =
```

```
    [ {I,vxv(Row,Col)} || {I,Row} <- Rows ],
  lists:filter( fun({_I,V}) -> V /= 0 end,
                DotProducts).
```

In the above definition, the computation of `DotProducts` is the element-wise application of the

```
fun ({I,Row}) -> {I,vxv(Row,Col)} end
```

function on `Rows` (representing the sparse matrix), where `Col` is a bound variable (bound to the sparse vector). Apart from some really contrived uses, a list comprehension denotes the element-wise processing of a list, and hence it is often a good candidate for the introduction of the task farm pattern. The previous section showed how a task farm can be introduced in the `mxv/2` function.

How is the task farm pattern implemented? The main idea is to introduce a given number of tasks (or processes, in Erlang terms), which are able to perform the "worker function" (the computation to apply on each element of the input), and keep them busy by providing them with items from the input. Typically, we have fewer tasks than input items. Once a task completes the computation on an input item, a new input item is assigned to it. Besides the worker tasks, two more tasks can be introduced, which organize the work: one is sending input items to idle tasks, and the other one is collecting results.

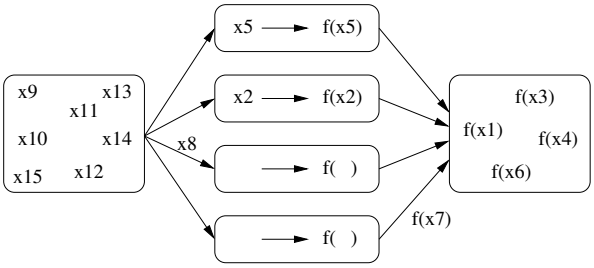


Figure 2: Task farm

There are two main use cases for the task farm pattern. In the first case, the input items are already available when the computations start. The example above belongs to this case: the input is a *list* of items (i.e. the rows in the matrix). In the second case, the input arrives *while* the computation is being performed. This is the stream-oriented approach, which can be manifested in Erlang with the application of a **`receive`**-expression. The input is made up of (possibly infinite number of) input items, which are processed as they become available.

As an example, consider a software component that receives integer numbers and factorizes them with a `factors/1` function. The prime factors of each received number are sent, together with the number, to some other process. The factorizer component terminates when (if) an `eos` atom is received. The standard implementation of the factorizer component is

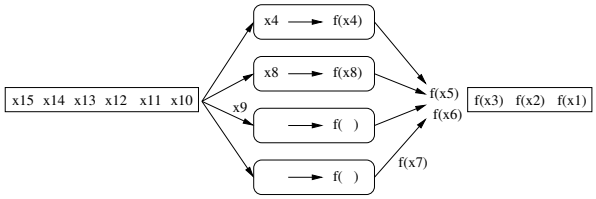


Figure 3: Order-preserving task farm

a tail-recursive function[2], which takes as an argument the *pid* of the process to which the output is to be sent.

```
factorize( CollectorId ) ->
  receive
    eos -> ok; % end of stream
    N -> CollectorId ! {N,factors(N)},
         factorize(CollectorId)
  end.
```

The component can be parallelized by the introduction of a task farm, which allows the `factors` function be applied on different input items in parallel. The `factorize/1` function is therefore a task farm candidate as well.

Note that the speedup that can be achieved by a task farm is bounded by the number of started worker tasks, and also by the number of available cores in the underlying hardware. For computation-bound workers it does not really make sense to have more worker tasks than cores. Also note that in order to approach the upper bound for the speedup the work should be distributed among the workers as evenly as possible.

3.2 Order-preserving task farm

The task farm pattern presented above is suitable in situations where we want to process an input *data set*, and produce a result *data set*. However, when the order of the items in the input and the output does matter, we must modify the pattern slightly.

In the factorization example, the output is tagged with the input value, the number `N` to be factored. The component which processes the prime factors is maybe independent of the order in which the numbers to be factorized were received. In fact, in many applications, it is enough to know which input data item produced which output data item, and the order in which the output is produced may not matter.

In the sparse matrix-vector multiplication example, however, we require that the output is a valid (sparse representation of a) vector, and hence the output tuples must be ordered by their first component, the index. The input is also ordered by (row) index, and hence we just need to produce the output in the very same order: the right pattern to apply is the *order-preserving* task farm.

If we compare Figure 2 and Figure 3, we can observe that in the case of the order-preserving task farm the output data items are not emitted (e.g. `f(x5)` and `f(x6)`) until all preceding items (i.e. `f(x4)`) are computed.

[2]Tail-recursive functions are optimized to loops in Erlang.

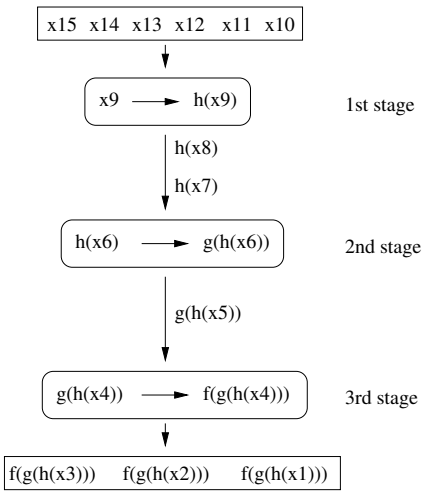


Figure 4: Pipeline

3.3 Pipeline

Our next parallel pattern, the pipeline, combines data parallelism with task parallelism. A pipeline consists of a sequence of stages, where each stage may perform a different computation, and the stages can be executed on different processors (or cores).

The pipeline pattern can be exploited very well when we want to compute a function composition on many data items. Let us assume that we have a sequence $x_1, x_2, \ldots x_n$ of data items, and for each x_i the value of $fghx_i$ is to be computed. We can construct a pipeline with three stages, each stage applying one of the three functions, as illustrated in Figure 4. An obvious candidate for this pattern may look like this.

```
[f(g(h(Item))) || Item <- Input]
```

A list comprehension with a function composition in the head is a good candidate for the pipeline pattern. However, there are many other forms of pipeline candidates. Consider, for example, `lists:map(fun d/1, Input)`, where `d/1` is defined as follows.

```
d(X) ->
  Y = a(X),
  Z = b(X),
  c(Y,Z).
```

One can easily refactor this second example into the first one by introducing the following definitions.

```
h(X)        -> {X, a(X)}.
g({X,AX})   -> {AX,b(X)}.
f({AX,BX}) -> c(AX,BX).
```

Note that `f(g(h(X)))` is equal to `d(X)` for any `X`. Therefore the expression `lists:map(fun d/1, Input)` is indeed a pipeline pattern candidate.

The pipeline pattern is most effective when the processors are all kept busy during the computation. In such a case the parallel speedup is close to the number of the pipeline stages. There is, however, some overhead to pay. When the pipeline processing starts, only the first stage is doing work. Then gradually the pipeline is filled, and all stages are working. In the end, there is again a falloff phase, when less and less stages are working, until the last output data item leaves the pipeline. When the number of data items provided to the pipeline is too small, the overhead due to the entering and falloff phases may be too high, and, together with other sources of overhead, may ruin speedup completely.

Another major source of performance degradation for a parallel pipeline is when the stages are not balanced: the slowest stage will determine the speed of the complete pipeline. In a large system, however, the processing units executing the faster stages may be utilized for some other computational activities, and hence their idle time is not wasted.

A good application area of a pipeline is when different stages are best executed on different types of hardware. For instance, we may introduce a 3-stage pipeline because the middle stage is best executed on a GPU, while the first and third stages can be run on CPU cores. Such heterogeneous (mixed CPU and GPU) patterns are of great practical importance, and they are supported by the Skel library as well.

3.4 Divide and conquer

The divide-and-conquer (*d&c*) pattern describes a computation where a problem is recursively divided into independent subproblems, and then the solutions of the subproblems are combined to produce the solution of the original problem.

Consider, for example, the following program to solve the famous *n-queens* problem[3]. A queen on a chessboard is represented as a pair of a row index and a column index. A queen *attacks* another one if they are in the same row, column or diagonal. The *legal* function checks if a queen does not attack any other queen in a list of queens.

```
attacks({RowA, ColA}, {RowB, ColB}) ->
   RowA == RowB orelse ColA == ColB orelse
      abs(RowA - RowB) == abs(ColA - ColB).

legal(Queen, Queens) ->
  lists:all(fun(Q) ->
              not (attacks(Queen, Q))
            end, Queens).

solve(N, Row, Queens) when Row > N ->
  [Queens];
solve(N, Row, Queens) ->
  % the following two properties must hold:
  % Row == 1 + length(Queens)
  % all( fun(I) -> element(1, nth(I,Queens)) == I
  %      end, seq(1,Row-1) )
  lists:flatmap(
    fun(Qs) -> solve(N,Row+1,Qs) end,
    [ [{Col,Row} | Queens]
      || Col <- lists:seq(1,N),
```

[3]In the n-queens problem, all possible non-attacking configurations of n queens on an $n \times n$ chessboard are to be found.

```
          legal({Col,Row},Queens) ]
    ).

queens(N) when N > 0 -> solve(N,1,[]).
```

The main computation is in the `solve/3` function, which extends a partial solution (`Queens`, a list of non-attacking queens on the board) with one more queen (in row with index `Row`) in all possible ways, and then recurses on these partial solutions (until the problem size, `N`, is reached).

We can observe that the `solve/3` function implements a divide-and-conquer behaviour: it creates multiple subproblems, calls itself recursively on these subproblems, and finally combines the solutions of the subproblems. To make this even more clear, we can rewrite the body of the second clause in the following way.

```
SubProblems =
  [ [{Col,Row} | Queens]
    || Col <- lists:seq(1,N),
       legal({Col,Row},Queens) ],
Solutions   =
  [ solve(N,Row+1,Qs) || Qs <- SubProblems],
lists:append(Solutions).
```

The main constituents of a *d&c* computation are:

- a function to divide a problem into subproblems;
- a predicate indicating that a problem need not be further divided (*base case* is reached);
- the computation to apply in the base case; and
- a function to combine the solutions of subproblems.

In our n-queens example, a problem is represented as a triple: the constant `N`, a row index, and a list of already placed queens. The division function adds a new non-attacking queen in the given row to the list of queens, and increases the row index by one. The base case predicate decides if the row index is greater than `N`. The function to be applied in the base case is very simple here: it wraps the list of queens with one more list constructor. Finally, the combining function is `lists:append/1`.

A divide-and-conquer computation is fairly easy to parallelize: we can work with the subproblems independently, in parallel. This is as simple as applying the task farm pattern whenever a problem is split into subproblems. A parallel *d&c* computation can be implemented by recursively starting task farms until the base case of the computation is reached. However, this naïve approach may yield too small tasks, too fine-grained parallelism. An efficient implementation may switch from parallel processing of subproblems to sequential when a certain condition holds (maybe several steps before the base case is reached). This condition may be related to the overall number of parallel tasks created by the computation, or to the size of a subproblem.

3.5 Composable patterns with algorithmic skeletons

The parallel patterns we have seen so far are already capable of describing the structure of simple parallel computations. However, more complex computations may require a combination of these patterns. A library for algorithmic skeletons may not only provide implementation for the afore-mentioned patterns, but it may also allow us to compose the skeletons in order to describe complex parallel computational structures. The Skel library is designed along these lines: parallel computations are expressed as nestable *workflows*.

To express a task farm in Skel, the following notation is used.

```
{farm, Workflow, NrWorkers}
```

The first component of this triple is the `farm` atom, the second is the workflow to be executed by the worker tasks on each input item, and the third component is the number of worker tasks in the farm. In the simplest case, the second component is a "sequential workflow", just as we have seen in Section 2.2:

```
Function =
  fun ({I,Row}) -> {I,vxv(Row,Col)} end,
... {farm,[{seq,Function}],length(Rows)} ...
```

The workflow `[{seq,Function}]` means here that the function `Function` will be applied in each worker task on incoming input items. This workflow contains a single workflow item, a pair of the atom `seq` and a fun-expression (of arity 1).

The pipeline pattern is expressed as `{pipe, Workflow}` in Skel, where the second component of the pair should be a list of workflow items, each executed in a pipeline stage: the number of pipeline stages is equal to the length of the `Workflow` list. An example of nesting of patterns is when a task farm is used as a stage in a pipeline.

```
{pipe, [{seq,Preprocessing},
        {farm,SomeComputation,16}]}
```

Another example of nesting was shown in Section 2.2, when an order-preserving task farm was implemented by nesting a task farm into an *ord*-skeleton.

```
{ord,[{farm,[{seq,Function}],length(Rows)}]}
```

The ord-skeleton is a pair of the atom `ord` and a workflow.

Once a workflow is constructed, it can be applied on a list of input data using the `skel:do/2` function.

```
skel:do(WorkFlow, InputData)
```

Similarly, a workflow can be applied on a stream of input with the very same `skel:do/2` function, but this time the second argument of the call should be the name of a (callback) module. This callback module must contain an `init/0`, a `next_input/1` and a `terminate/1` function with the following role. The `init/0` function produces an initial state, which will be passed as an argument to `next_input/1`, which produces the next input item from the stream, and updates the state. When the stream is exhausted, `next_input/1` must return the `eos` atom. Finally, the `terminate/1` function will be applied on the final state.

This process is demonstrated on the `factorize/1` example, which can be rewritten to a task farm in the following way. (No state is required in this example, therefore the atom `no_state` is used as a dummy.)

```
factorize( CollectorId ) ->
  Fun = fun(N) ->
          CollectorId ! {N,factors(N)}
        end,
  % use 16 workers in the task farm
  skel:do( [{farm, [{seq, Fun}], 16}], ?MODULE ).

init() -> {ok, no_state}.

next_input( no_state ) ->
   receive
       eos -> {eos,no_state};
       N   -> {input,N,no_state}
   end.

terminate( no_state ) -> ok.
```

Note that here we chose the current module as the callback module, the name of which is exposed by the predefined `?MODULE` macro.

Compared to the afore-mentioned algorithmic skeletons, the divide-and-conquer pattern is expressed quite differently in Skel. This pattern is made available as a set of higher-order functions in the module `sk_hlp` (which stands for *Skel high-level patterns*). The `sk_hlp:dc/4` function takes four functions as arguments (`IsBaseCase`, `BaseFunction`, `Divide` and `Combine`), and returns a function, which accepts a problem, and produces the solutions using a parallel divide-and-conquer method. For example, the n-queens problem can be implemented in the following way.

```
queens(N) when N > 0 ->
  Problem = {N,1,[]},% meaning {N,Row,Queens}
  IsBaseCase =
    fun({N, Row, _Queens}) -> Row > N end,
  BaseFunction =
    fun({_N, _Row, Queens}) -> [Queens] end,
  Divide =
    fun({N, Row, Queens}) ->
      [ {N, Row+1, [{Col,Row} | Queens]}
        || Col <- lists:seq(1,N),
           legal({Col,Row},Queens)
      ]
    end,
  Combine = fun lists:append/1,
  sk_hlp:dc(IsBaseCase,
            BaseFunction,
            Divide,
            Combine)(Problem).
```

This solution will solve each subproblem in a new task. To limit the granularity of parallelism, `sk_hlp` offers the `dc/5` function, which takes one more parameter: the number of tasks to be created during the computation.

3.6 Component hygiene

When a sequential computation is refactored into parallel, care must be taken to preserve the functionality of the code. In a pure functional language functions are free of side-effects. In such a language the confluence of computation is guaranteed: evaluation order of subcomputations has no effect on the result (although it may affect termination). Even the parallel evaluation of subcomputations will yield the same result. Therefore, it is fairly safe to introduce parallelism into pure computations.

In an impure language, like Erlang, expressions may have side effects, and hence neither the evaluation order of subexpressions can be freely changed, nor sequential evaluation can be freely turned to parallel. It is still safe to introduce parallelism if we restrict ourselves to parallelizing only pure computations. In this case all workflow items are pure functions, and are safe to execute in parallel.

In Erlang, however, pure functions are rather rare [42]. In order to have useful patterns, we need to weaken the purity requirement on workflow items, otherwise no real code could be parallelized. For this reason we allow impure computations in workflows, but only if the side effects they produce do not interfere with each other. Such workflow items are called "hygienic components", and are explained in more details in [6].

4 TOOL SUPPORT FOR PARALLELISATION

The ParaPhrase Refactoring Tool for Erlang (PaRTE) combines parallel pattern discovery, candidate prioritization and semi-automated refactoring for shaping Erlang code and introducing the algorithmic skeletons provided by the Skel library. This section presents the methodology supported by PaRTE, and explains how to use this tool.

4.1 Installing PaRTE

PaRTE is an open-source tool built upon the static source code analysis and transformation tool RefactorErl [19]. The source code of PaRTE can be downloaded from its wiki page [20] and can be compiled on Linux-based systems.

PaRTE has two fundamental software dependencies:

- a complete Erlang system (Erlang OTP 17 or higher) and
- a C++ compiler (e.g. *g++ 4.5* or higher).

The steps to install and to configure PaRTE are the following:

- download the source code bundle of the tool;
- unzip it in a directory of your convenience;
- open a terminal and `cd` to the directory of PaRTE;
- type "`./install_parte -build parte`" to start the building and installation process.

Many Erlang developers prefer to use Emacs as an integrated development environment. This is the reason why the main user interface of PaRTE is also an Emacs plug-in. To enable PaRTE in Emacs, some lines should be added to the Emacs configuration file *.emacs*, which is located in the user's home directory. The lines to be added are displayed by the installation script.

4.2 Pattern candidate discovery with PaRTE

The PaRTE tool is written (mostly) in Erlang, and hence it can be conveniently operated from an Erlang shell. Later in this section the Emacs integration will also be explained, but at first we shall see the hardcore Erlang developer's view of the tool.

To launch the tool, execute the following shell command from the PaRTE installation directory.

```
referl/tool/bin/referl -db nif -base referl/tool
```

This command starts an Erlang VM, and inside the VM it configures and starts PaRTE (as an Erlang/OTP application). The core component of PaRTE is a RefactorErl instance – that is why `referl` appears in the above shell command. The name of the started Erlang VM is `refactorerl@localhost`. Note that it is possible to use all the source code analysis and refactoring features of RefactorErl from within PaRTE.

The parallel pattern candidate discovery feature of PaRTE is accessible through the interface modules `ri` and `refpp_api`. We can call the exported functions of these modules from the command shell (REPL) of the started Erlang VM. When we start to work with some Erlang code base, the first thing to do is to load the relevant source files into the tool. We can do this file by file:

```
ri:add("path_to_my_file/my_file.erl").
```

but we can also load the contents of a whole directory:

```
ri:addenv(include, "path_to_my_headers").
ri:add("path_to_my_files").
```

or a complete application as well:

```
ri:addenv(appbase,"path_to_my_app").
ri:add("path_to_my_app", "app_name").
```

The following command starts pattern candidate discovery on the loaded code base, and returns a list of found candidates.

```
refpp_api:find().
```

The result of this function, a data structure containing detailed information on the identified pattern candidates is useful for further processing; however, an end-user may prefer to call the `run/0` function instead, which pops up a web-browser, and displays the discovery results in it. This can be seen on Figure 5. The information presented about a candidate comprises the kind of the pattern (e.g. Δ refers to a task farm, || to a pipeline, and ⋄ to a *d&c* candidate), the location of the candidate (module, function/arity), and an estimation on the effect of parallelization. PaRTE attempts to measure the (sequential) execution time of the pattern candidate, and to provide an estimation on expected parallel speedup using a cost model [9], which takes into account the number of processor cores, as well as the overhead of spawning processes and that of inter-process communication. This speedup estimation technique [7] extracts a pattern candidate expression into a separate source code, generates a series of random input to it, compiles and executes the code, and measures the execution time. This technique is applicable when PaRTE is able to infer the type of free variables in the pattern candidate, and to generate meaningful random values as input. When this is not possible, PaRTE cannot provide a speedup estimation, but computes a computation intensity metric for the pattern candidate instead. The larger the value of this metric is the more likely that parallelization will improve performance. The metric takes into account the number of basic operations, and iteration structures such as recursion and list comprehensions. Either speedup estimations or computation intensity metrics are computed, PaRTE can prioritize identified pattern candidates, and present the user the most promising candidates in terms of potential gains in execution time.

To customize pattern discovery, one can provide a "property list" as an argument to the `run/1` function. The following call will discover all task farm and pipeline candidates in module `my_file` excluding nested patterns.

```
refpp_api:run([{modules, [my_file]},
               {cand_types, [farm, pipe]},
               {chains, true}]).
```

There are four customizable properties that can be passed to `refpp_api:run/1`.

`modules`: The name of the modules (as list of atoms) in which pattern candidate discovery should take place. (Default value: all loaded modules.)

`cand_types`: The types of pattern candidates to search for. A list from the following values can be used: `farm`, `pipe`, `pool` and `dnc`. (Default value: all four types of candidates.)

`chains`: A boolean value – whether to discover candidates of compound (nested) patterns. (Default value: false.)

`rewrite`: A boolean value – whether to apply *skeleton rewrite rules*[4] on the candidate discovery results. (Default value: false.)

4.3 Automated parallelization transformations

Besides pattern discovery, PaRTE is able to automatically transform Erlang source code, and to replace a pattern candidate with a Skel-based implementation of a parallel pattern [6]. Let us demonstrate this through the source code of an open-source implementation [46] of multi-agent systems.

We have downloaded and unzipped the project under the `/home/X/mas` directory. The first thing to do after launching PaRTE is to set the include path, and to load the source files to work with into the tool. For the sake of simplicity, we load only one source file from the project: `misc_utils.erl`. The next step is to start pattern candidate discovery.

```
ri:addenv(include,"/home/X/mas/include").
ri:add("/home/X/mas/src/utils/mas_misc_util.erl").
refpp_api:run().
```

[4] Nested patterns can often be rewritten into other nested patterns using a set of well-known rules [1]. These rules express language-independent structural equivalence of compound algorithmic skeletons. For example, a task farm of pipelines can be rewritten to a pipeline of task farms.

Pattern Candidate Browser

Transformation sequences

ID	Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)	Metric	Recommended?
1	(Δe45)	matrix	dotproduct	2	2	1.76	15	✔
2	(Δe25)	matrix	multiply	2	2	1.75	27	✔
3	(Δe21)	matrix	multiply	2	2	1.74	18	✔

Figure 5: Pattern candidates found by `refpp_api:run/0`.

Pattern Candidate Browser

Transformation sequences

ID	Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)	Metric	Recommended?
5	(oe501)	mas_misc_util	count_funstats	2			120	✔
1	(Δf9)	mas_misc_util	count_funstats	2			120	✔
4	(Δe154)	mas_misc_util	generate_population	2			16	
3	(Δe195)	mas_misc_util	meeting_proxy	4			16	
2	(Δe497)	mas_misc_util	count_funstats	2			14	

Chart options

Apply selected transformations

Details of the transformation sequence

Configuration	Location information	Program text	Number of workers	Sequential CPU time	Parallel CPU time	Expe
(Δf9)	/home/guest/mas/src /utils/mas_misc_util.erl : {{125,1},{125,14}} - {{132,67}, {132,67}}	count_funstats(_,[]) -> []; count_funstats(Agents, [{Stat, MapFun, ReduceFun, OldAcc}|T]) -> NewAcc = lists:foldl(ReduceFun, OldAcc, [MapFun(Agent) || Agent <- Agents]), [{Stat, MapFun, ReduceFun, NewAcc} | count_funstats(Agents,T)].				

Figure 6: Pattern Candidate Browser – detailed view.

PaRTE shows the result of pattern candidate discovery in a web browser. We can see that the tool has found four task farm and one feedback candidates in the `mas_misc_utils` module. In this case speedup estimations were not possible to obtain, therefore the candidates are ranked based on the computation intensity metric. In order to get more detailed information about a particular pattern candidate, the user can click on a row in the table (Figure 6). The detailed view contains the text (source code) of the candidate, and – if applicable – some information about the parallel speedup estimation.

Based on the position information presented in the detailed view, the user can easily find the candidate in their favorite editor, for example in Emacs (Figure 7). In this case, the task farm candidate appears in the `count_funstats/2` function, which calculates different statistics over a list of agents. The second argument, as well as the result of the function, is a list of "`funstats`", as described by the **`spec`** line in the source

code.[5] Here `funstat` can be considered as a user defined type name for a tuple with four elements.

We can observe that the `count_funstats/2` function processes the input `funstat` list elementwise. The recursion structure of this function is identical to that of the standard `lists:map/2` function: for each `funstat` in the input an updated `funstat` is computed in the output. The update concerns the fourth element in the `funstat` tuple: the original value `OldAcc` is replaced by a new value `NewAcc` (computed with a map-reduce computation). A map-like function, such as `count_funstats/2` can easily be transformed into a task farm, this is why PaRTE identified this function as a pattern candidate.

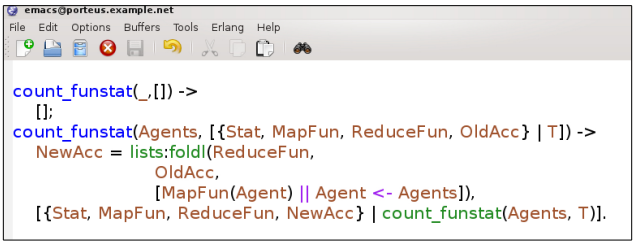


Figure 7: Source code of a candidate in Emacs.

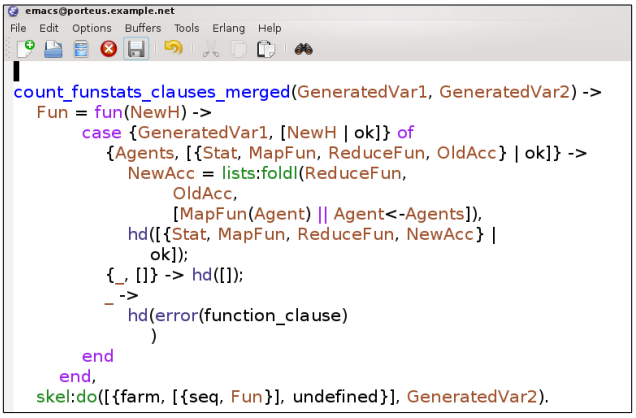


Figure 8: Transformed source code in Emacs.

PaRTE is able to automatically introduce the parallel pattern for an identified pattern candidate. The user has to select the pattern candidate on the web interface, and press the *Apply selected transformation* button (Figure 6). After asking for confirmation, the tool performs the transformation. (Note that the web interface does not show the refactored source code.) The user can check the result of the transformation by, for example, opening the modified file in Emacs (Figure 8).

The source code after the transformation seems to be overly complicated. This is the result of the quite generic rewrite scheme the transformation is based upon [6]. However, the obtained code can be cleaned up substantially. Some clean-up transformations have already been implemented in PaRTE, but the ones required in this particular case are not yet. Therefore, unfortunately, we must clean up the code manually.

Consider the generated *kernel function* for the task farm, i.e. the fun-expression assigned to variable `Fun`. Its body is a case-expression with three clauses. Since the pattern of the first clause always matches, the case-expression together with its second and third clauses can be eliminated. Moreover, the call of the `hd/1` function on a head-tail expression in the first clause of the case-expression can also be simplified to the head expression.[6] After clean-up, the kernel function of the task farm becomes the following.

```
Fun =
  fun(NewH) ->
    NewAcc = lists:foldl(       ReduceFun,
                                   OldAcc,
          [MapFun(Agent) || Agent<-Agents] ),
    {Stat, MapFun, ReduceFun, NewAcc}
  end
```

The same code fragment can be parallelized in several different ways, and hence it can be identified as a candidate for different patterns. On Figure 9 the introduction of another parallel pattern for `count_funstats/2` is presented. Here the list comprehension expression was identified as a candidate for, and replaced by, a task farm.

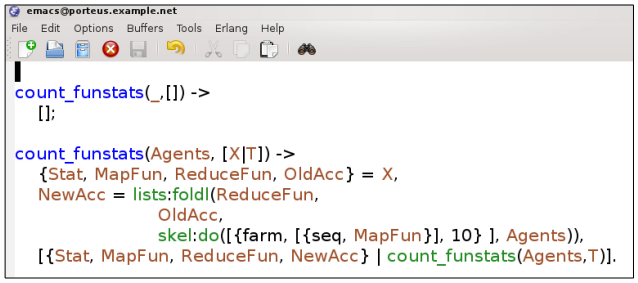


Figure 9: Another parallelization of `count_funstats/2`.

4.4 Step-by-step and manual transformations

PaRTE can automatically transform many pattern candidates into an application of a parallel pattern, but not all of those which can be identified by the pattern candidate discovery process. Most notably, the tool can find candidates for *d&c* and pool patterns, but is not yet capable of transforming them.

Consider the sequential implementation of the *n-queens* algorithm from Section 3.4, which contains function `solve/3`,

[5] Such type specifications for functions are often used for documentation purposes in Erlang, but they also serve as input to bug detection tools such as Dialyzer [47].

[6] Erlang supports *improper lists*, which can be constructed with the *head-tail* notation, such as `[1|ok]`, with the tail not being a proper list. In this particular case, the head of the improper list is the number `1`, and its "tail" is the atom `ok`.

a candidate for the *d&c* pattern. Assuming that the source code is located in the `n_queens` module, we can load it in the tool and run pattern candidate discovery for specifically targeting at divide-and-conquer candidates (with the appropriate options for function `run/1`).

```
ri:add("/home/guest/n_queens.erl").
refpp_api:run([{cand_types, [dnc]}]).
```

The result of the analysis can be seen on Figure 10. If we try to perform the automated transformation on the pattern candidate by pressing the *Apply selected transformations* button, PaRTE notifies us (in the shell) that this is not feasible. However, there is a technique [27] to refactor divide-and-conquer candidates using a sequence of small transformations using PaRTE, and arrive at the code presented in Figure 11. According to this technique, the user can define:

- the `is_base/1` function to provide the condition of a base case selection;
- the `base/1` function to provide the operation performed in base cases;
- the `divide/1` function to divide a problem into smaller subproblems; and
- the `combine/1` function to combine solutions of subproblems into a solution of a compound problem.

Then these functions can be passed as arguments to the higher-order `sk_hlp:dc/4` function, which is the Skel implementation of the parallel *d&c* pattern.

There are other pattern candidates which PaRTE can discover, but without built-in support for automatic or step-by-step transformation. Those candidates can be refactored into parallel patterns manually.

5 HOW TO BUILD A TOOL LIKE PARTE?

The ParaPhrase Refactoring Tool for Erlang is able to discover pattern candidates in Erlang source code, and to transform (many of) them into applications of parallel patterns. PaRTE builds upon RefactorErl, a static source code analysis and transformation framework. RefactorErl provides a huge variety of static semantic analyses, which are necessary for the identification of pattern candidates, as well as an interface to perform the necessary source code transformations. The tool takes a set of Erlang source files as input, and constructs a so-called *Semantic Program Graph* (SPG) from them, which can be persisted in a database for posterior processing. The SPG is a three-layered data structure containing lexical, syntactic and static semantic information about the code, and can be considered as a generalizaton of an abstract syntax tree. Semantic information about variable scoping, the function call graph etc. are represented as semantic nodes and edges in the SPG.

The goal of pattern candidate discovery is to find code fragments which exhibit a specific structure and behaviour. The candidate may take many syntactic forms, and the discovery attempts to recognize most of them. Pattern candidates are characterized as the various syntactic forms plus further properties in terms of control-flow, data-flow, and data-dependence information.

5.1 Supporting analyses

First of all, we shall briefly cover the interpretation of standard analyses, such as control-flow and data-flow analyses, in a functional language like Erlang [45]. Then we can turn our attention to the more specific analyses, those related to the discovery of candidates of various patterns.

As a general note, we must emphasize the difference between zeroth-order, first-order and higher order analyses. The zeroth-order analyses are performed on an abstract syntax tree, and are typically over-approximations – higher order analyses take more context information into account (i.e. already collected semantic information), and hence yield more precise analysis results. The key trick is that the results of one analysis can be used to refine the input for another. Therefore, one can execute an analysis sequence repeatedly – theoretically, until a fixed point is reached. This way higher order analysis results are obtained. Since the iterative execution of all analyses is definitely expensive for large programs, in practice first-order analyses are already considered good enough.

5.1.1 Control-flow analysis. In a functional programming language, the execution of a program is the evaluation of its expressions. The control-flow analysis computes the evaluation order of expressions. In a language with strict evaluation, like Erlang, defining evaluation order is more straightforward than in one with lazy evaluation.

The control-flow analysis is performed from a given entry point of a program. In Erlang, there is no designated entry point: any exported function (a function that is visible from outside of the defining module) can act as an entry point.

The *inter-procedural control-flow graph* of an Erlang program with entry point f is denoted as $G_{CF}f$. This is a directed graph with labeled edges. Briefly, the nodes of the graph are the nodes in the syntax tree plus some auxiliary nodes associated to function calls, as defined below. The edges represent evaluation order, and the labels on the edges are conditions. The graph can be constructed from the *intra-procedural control-flow graphs* of the functions defined in the program.

The intra-procedural control-flow graph for a function can be built up by visiting its abstract syntax tree. The nodes corresponding to the expressions in the body of the function are connected in post-order ("subexpressions first"). Branching expressions induce nodes with multiple outgoing edges. The labels attached to such edges are the appropriate conditions. (All other edge labels are *true*.)

For each function g two auxiliary nodes are created, which serve as source and sink nodes of the intra-procedural control-flow graph: $start_g$ and end_g. There is an edge from $start_g$ to the first pattern of the first function clause of g, and there are edges from each return expression of g to end_g. (If we regard the intra-procedural control-flow graph as a partial

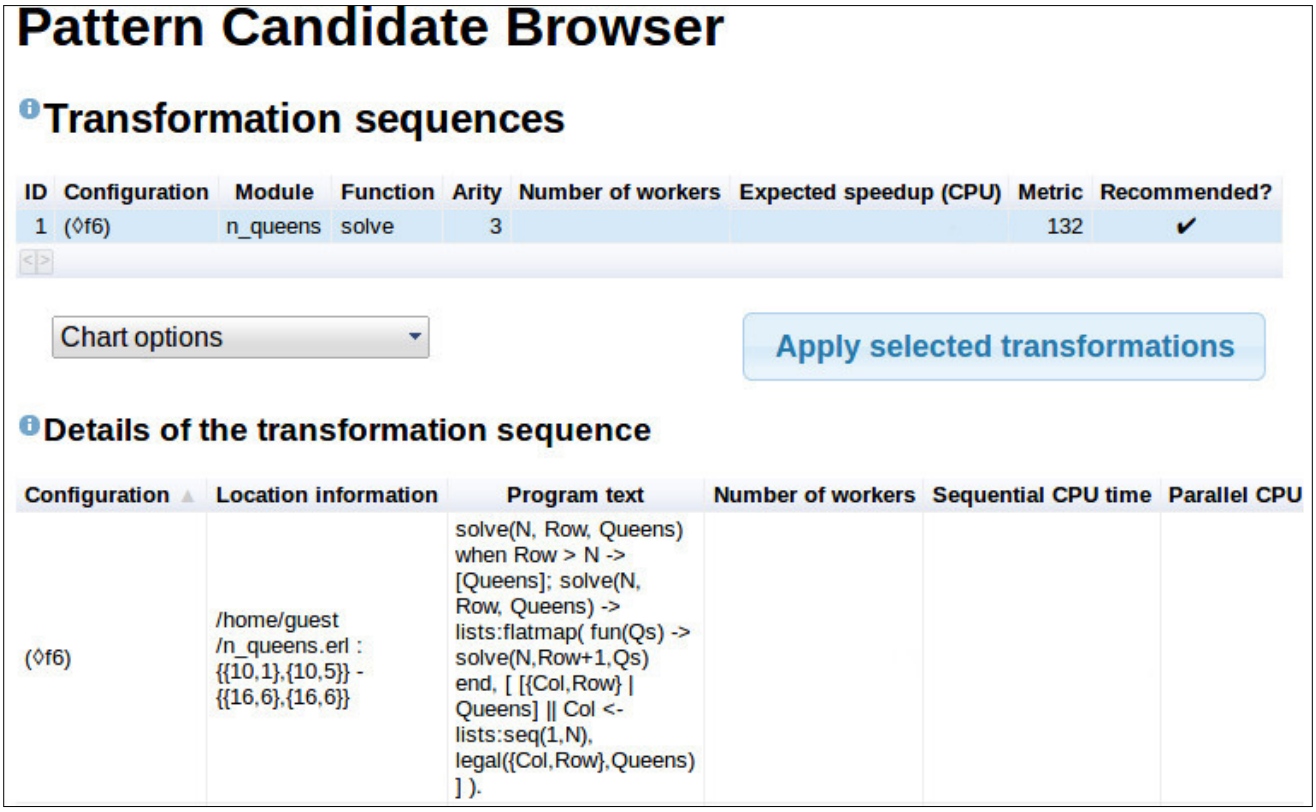


Figure 10: *d&c* candidate – detailed view

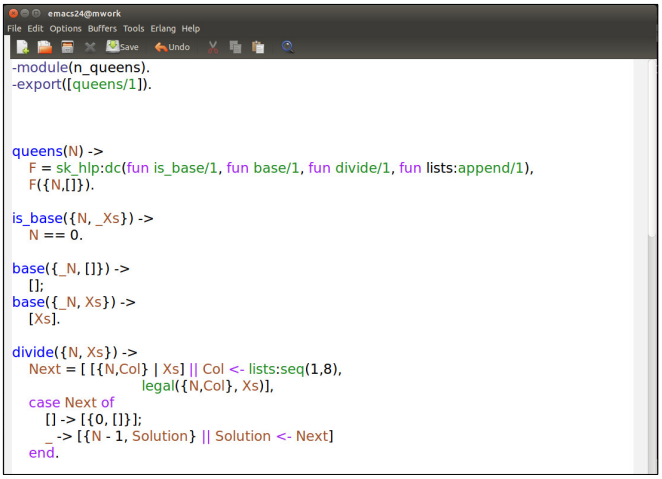


Figure 11: *d&c* pattern instance

order over the constituting expressions, $start_g$ is the bottom, and end_g is the top value of the partial order.)

Function calls will also be represented with auxiliary nodes. These nodes will connect the intra-procedural control-flow graphs to form the inter-procedural control-flow graph. If a call to a function h occurs in any of the intra-procedural control-flow graphs (including that of h), a graph node c representing the call is replaced with two new nodes: $call_h^c$ and ret_h^c. All incoming edges a, c (for some other node a) are replaced with edges $a, call_h^c$, and all outgoing edges c, a are replaced with edges ret_h^c, a. Moreover, the edges $call_h^c, start_h$ and end_h, ret_h^c are also added to $G_{CF}f$.

There is one more important concept related to the control-flow graph, namely execution paths. An execution path is a path in $G_{CF}f$, which may visit an edge multiple times. A finite execution path terminates with the end_f node, but infinite execution paths are also possible. Given a $G_{CF}f$, the set of execution paths starting from a node v will be denoted by EPv.

5.1.2 Data-flow analysis. Similarly to the inter-procedural control-flow graph, the *data-flow graph* G_{DF} of an Erlang program can be defined as a directed graph with labeled edges. Again, the nodes are the nodes of the AST – representing the (sub)expressions of the analyzed program.

The graph edges describe the flow of data. An edge from u to v represents the fact that there may be an execution of the program where the value of the expression u flows into expression v. Labels on the edges characterize the different types of data flow.

- If the value of u provides the value for v, the *flow* label is used. For example, the actual parameter of a function *flow*s into the formal parameter, or the right-hand side of a match expression *flow*s into the left-hand side expression. For example, in the expression

```
{A,B} = {X+Y,Y}
```

  the expression `{X+Y,Y}` *flow*s into `{A,B}`.
- If v is a tuple, list etc. expression, and u provides the value of a substructure (i.e. i^{th} element of a tuple/list, the tail of a list etc.), the label is *construct* – more specifically, it is $construct_i$, $construct_{tail}$ etc. For the match expression above, for example, there exists a $construct_1$ edge from `X+Y` to `{X+Y,Y}`.
- If u is a tuple, list etc. expression, and v extracts a component, then the label *select* is used, e.g. $select_i$, $select_{tail}$ etc. For the match expression above, for example, there exists a $select_1$ edge from `{A,B}` to `A`.
- Other data flow dependencies between expressions are labeled with *dep*, like the edge from `X` to `X+Y` in the example above.

From the *flow*, *construct* and *select* edges of G_{DF}, the *data-flow reaching* relation can be computed. The computation of *data-flow reaching* pairs matching *construct* and *select* edges. For example, in the match expression above the expression `A` is reachable from `X+Y`, because of the $construct_1$ – *flow* – $select_1$ path in the G_{DF}. More formally, the zeroth-order data-flow reaching relation ($\overset{\mathbf{0f}}{\rightsquigarrow}$) is the minimal relation which satisfies the following rules.

$$n \overset{\mathbf{0f}}{\rightsquigarrow} n \qquad \text{(reflexive)}$$

$$\frac{n_1 \xrightarrow{\mathbf{flow}} n_2}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2} \qquad \text{(flow rule)}$$

$$\frac{n_1 \xrightarrow{\mathbf{construct_i}} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3,\ n_3 \xrightarrow{\mathbf{select_i}} n_4}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_4} \qquad \text{(c-s rule)}$$

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_3} \qquad \text{(transitive)}$$

The third rule expresses that once we pack a value into a tuple/list, and the value of the tuple/list reaches some expression where we unpack the same element of the tuple/list, then we can say that the original value reaches the unpacked value.

A detailed and more precise definition of the data-flow graph and the reaching relation can be found in [45], where the context dependent *first-order data-flow reaching* is also explained. The so-called *compact data-flow reaching* relation is defined as a subset of the data-flow reaching relation. In zeroth-order the following rule gives its definition.

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\quad \nexists n_0.\ n_0 \neq n_1 \wedge\ n_0 \overset{\mathbf{0f}}{\rightsquigarrow} n_1}{n_1 \overset{\mathbf{0f_c}}{\rightsquigarrow} n_2} \qquad \text{(compact)}$$

5.1.3 Derived information.

Function call graph. Being a functional language, Erlang supports higher-order functions. Therefore, the *function call* analysis depends on the result of the data-flow analysis (and vice-versa). One can perform higher order analyses by running multiple passes of these analyses, each using the results of previous passes. As mentioned above, an approximation can be achieved by applying a fixed (but customizable) number of iterations. The result of the function call analysis is the function call graph, which will be denoted as G_{FC}.

Dependence graph. The control dependence graph can be computed from the control flow graph as described in e.g. [39, 40]. Then this graph is further extended with the *dep*-edges of the G_{DF} and the data-flow reaching relation to form G_Df, the *dependence graph* of the program with entry point f. When a *dep*-edge or a data-flow reaching edge is to be added to the G_Df from, or to, a function call expression, then outgoing edges will start from the respective *ret* node of the control-flow graph, and incoming edges will arrive at the *call* node. A path from u to v in G_Df will be denoted by $u \overset{\mathbf{dep}}{\rightsquigarrow} v$.

5.2 List comprehensions as pattern candidates

List comprehensions express the repeated evaluation of an arbitrary expression over the elements of a given list. If this expression is free of side-effects, or at least hygienic (Section 3.6), the list comprehension can be regarded as a candidate for task farm. Furthermore, if the expression is a composition of two or more functions, or it is a function that calls other functions in its body, then it can be considered as a pipe candidate as well. For example, the list comprehension

```
[foo(X) || X <- List]
```

can be transformed to the following Skel call.

```
skel:do([{farm, fun foo/1, 10}], List)
```

When the source contains more than one generators and filters, some shaping is required first. One possible technique is to pre-generate an intermediate data structure. Consider, for example, the list comprehension

```
[foo(X, Y) || X <- List1, Y <- List2]
```

which can be refactored first into

```
List3 = [{X, Y} || X <- List1, Y <- List2],
[foo(X, Y) || {X, Y} <- List3]
```

and then Skel is introduced in the second step.

```
List3 = [{X, Y} || X <- List1, Y <- List2],
skel:do( [{farm, fun({X,Y}) -> foo(X, Y) end, 8}],
         List3 )
```

This technique increases the memory usage of the code, which might be significant if the original list comprehension generates a long list.

5.3 Recognizing certain library calls

The Erlang standard library defines several functions which have obvious parallel counterparts. Most notably, consider `lists:map/2`, the call of which can be categorized as a task farm or a pipeline candidate. It is worthwhile to build knowledge specific to standard libraries into the refactoring tool, therefore we have manually analyzed some commonly used Erlang/OTP modules.

- Modules for list manipulation: `lists`, `proplists`, `ordsets`, `orddict`.
- Modules for manipulating other data types: `array`, `sets`, `dict`, `gb_trees`, `gb_sets`. The functions in these modules operate on data which can be transformed to and from lists. This allows us to introduce a parallel pattern on a temporary list data structure at the price of two data conversions and the extra memory used by the temporary data structures. Parallel skeletons implemented specifically for arrays, sets etc. would eliminate this performance overhead.
- Modules for manipulating *term storages* and databases: `ets`, `dets`, `mnesia`. The functions in these modules also manipulate collections of data, but they have side-effects. Hence the safe introduction of parallel patterns may require deeper analysis regarding component hygiene.

Pattern candidate discovery involves the search for the application of certain functions in the above-mentioned modules.

Module `lists`. This module defines the standard list manipulation functions, many of which can be given an alternative, parallel implementation. The applications of `map/2`, `filter/2`, `foreach/2`, `zipwith/3`, `zipwith3/4` and `flatmap/2` are recognized as candidates of the task farm (and possibly the pipeline) pattern. Moreover, the applications of `foldl/3`, `foldr/3`, `all/2`, `any/2`, `max/1`, `min/1` and `sum/1` are recognized as candidates of the reduce pattern.

Module `proplists`. This module defines functions operating on lists of tuples, where the first element of each such tuple is a key for lookups. The functions with a name starting with `get-`, `lookup-` and `substitute-` can be understood as filters and maps on lists, hence the applications of these functions are candidates of the task farm pattern. We have to note here that these operations, in most of the cases, are not complex enough to be worthwhile to execute in parallel. A suitable cost model can help in the identification of promising candidates.

Modules `ordsets` and `sets`. These modules are collections of functions for manipulating sets. While ordsets are represented as ordered lists by definition, the representation of sets is unspecified. Sets can be easily converted to and from lists with functions `to_list` and `from_list`. The applications of `filter` are candidates of the task farm pattern, and applications of `fold`, `intersection` and `union` are candidates of the reduce pattern.

Modules `orddict` and `dict`. The functions in these modules manipulate dictionaries of key-value pairs. Similarly to the previous case, orddicts are represented as ordered lists, and the representation of dict is unspecified – the latter being convertible to/from lists (with functions `to_list` and `from_list`). The applications of `map`, `filter` and the various `update...` functions can be recognized as task farm, and those of `fold` can be regarded as reduce pattern candidates.

Module `array`. This module defines functions on extendible arrays. The representation of arrays is unspecified, but there are conversion functions `to_list` and `from_list` for this construct too. The applications of `map` and `sparse_map` are taken as candidates of the task farm pattern. Applications of `foldr`, `foldl`, `sparse_foldr` and `sparse_foldl` are candidates of the reduce pattern.

Modules `gb_trees` and `gb_sets`. These modules define functions for balanced trees and ordered sets. Data is represented with deeply nested tuples. Therefore it would be possible to introduce data specific parallelism with tuple-specific extensions of the existing skeletons. However, the previously introduced method can also be applied: the structures can be converted to lists with `to_list`, and the skeletons can be introduced for these lists. For gb_sets the reconstruction from lists is straightforward, it can be performed with function `from_list`. The reconstruction of balanced trees is slightly more tricky. First we can construct an orddict from the list representation, and then we can transform the orddict to tree representation with function `from_orddict`. The interesting function applications from these two modules are those of `map` and `fold`, which can be recognized as task farm and reduce candidates, respectively.

Modules `ets`, `dets` and `mnesia`. These modules support data storage and shared data manipulations. As noted already, the functions in these modules have side effects, hence an appropriate side-condition check must be performed before applying any refactoring transformations on calls to these functions. All three modules define the `foldl` and `foldr` functions, which are candidates of the reduce pattern. Moreover, the `mnesia` module defines the `transform_table` function, which applies a function on the elements of a table, bearing a similarity to `lists:map/2`. The applications of this function are hence candidates of the task farm pattern.

5.4 List-based recursive functions

Farm and map patterns model embarrassingly stream/data parallel computation where a worker function is iterated over the elements of the input. Therefore, in the pattern discovery process we have to identify language constructs that express a form of iteration either over the elements of a data structure (i.e. list) or an input stream (i.e. received messages).

The previous section gave an overview of the standard library functions amenable to pattern based parallelization.

Many of these functions are higher-order, and can be parameterized to address a wide range of possible needs. However, these functions, or special cases thereof, are often reimplemented by (mediocre) programmers. This section provides a characterization of recursive functions which could also be written using the higher-order function `lists:map/2`, and hence are suitable candidates for a task farm pattern. In what follows, these functions will be referred to as *map-like functions*.

5.4.1 Map-like functions which are not tail-recursive. A programmer can write a function that processes a list elementwise in various ways, e.g. using list comprehensions, standard library functions (like `lists:map/2`) or defining a map-like recursive function. Their common property is that one can exploit data parallelism by transforming them into task farms.

Here we propose a set of conditions to be used for the identification of map-like functions. What are the main characteristics of a map-like function f? It should have a list parameter, and return a list. The head of the returned list may depend on the head of the input list, but may not depend on its tail. Similarly, the tail of the returned list may not depend on the head of the input list: it should simply be the result of a recursive call to f on the tail of the input list. The function may have additional parameters, but all these parameters must be passed to the recursive call unchanged.

```
f(Parameter,List) ->
  case List of
    [] -> [];
    [Head|Tail] ->
      X = ... Head ...,
      [X | f(Parameter,Tail)]
  end.
```

There are many syntactic forms that satisfy these requirements, one is shown above. Note that f need not call itself directly: we may identify a set of mutually recursive functions as map-like too.

Now let us investigate these requirements in a more formal way. We propose conservative rules to use for identifying map-like functions. Consider a function f. We will rely on the control flow and data flow graphs presented in section 5.1 built using f as the entry point, such as $G_{CF}f$. Thanks to the inter-procedural analyses, mutually recursive functions are also handled properly. The rules are the following.

(1) f must be recursive: the inter-procedural control-flow graph must contain a directed path from the starting node of f to a *call*-node of f.

$$\exists p \in EPstart_f \text{ and } \exists c \text{ such that } call_f^c \in p$$

(2) The definition of f must have a base case: there is a directed path from the *start*-node of f to the *end*-node of f which does not include a *call*-node of f.

$$\exists p \in EPstart_f : \big(end_f \in p\big) \wedge \big(\nexists c : call_f^c \in p\big)$$

(3) f should not be of divide-and-conquer style, in the sense that its definition should not recure multiple times "in breadth": on each execution path, after a recursive call to f has returned, no new call to f may occur.

$$\forall c_1 \text{ and } \forall p \in EPret_f^{c_1} : \big(\nexists c_2 : call_f^{c_2} \in p\big)$$

(4) f may have multiple parameters, among which there is one, a list, which will be processed element-by-element. For the sake of simplicity, we may now assume that the definition of f is provided with a single clause. (If f is defined with multiple clauses, we can meld those into a single clause by introducing an artificial **case**-expression that contains the pattern matching and the guards of the original function clauses.) Now we can expect that the head of the function clause is in the form of $f\overline{V_1}, \mathtt{L}, \overline{V_2}$, where parameter `L` corresponds to the list consumed by f.
We can define the semantic functions Head`L` and Tail`L` based on the data-flow graph as follows.

$$\text{Head}\mathtt{L} = \{n \mid \exists n' : \mathtt{L} \overset{\mathbf{Of}}{\rightsquigarrow} n',\ n' \xrightarrow{\mathbf{select_{head}}} n\}$$

$$\text{Tail}\mathtt{L} = \{n \mid \exists n' : \mathtt{L} \overset{\mathbf{Of}}{\rightsquigarrow} n',\ n' \xrightarrow{\mathbf{select_{tail}}} n\}$$

(5) The expression returned by f must satisfy further properties. Regarding the different execution paths of f, we can define the set R of return expressions by collecting the last evaluated expression on all paths before reaching the *end*-node of f. The last evaluated expression is connected to the *end*-node by an edge in $G_{CF}f$.

$$R = \{\varrho \mid \varrho, end_f \in G_{CF}f\}$$

For all return expression $\varrho \in R$, the following requirements must hold.

(a) If ϱ is on a non-recursive execution path, then ϱ must evaluate to an empty list, i.e. only an empty list constructor is in the compact data-flow reaching relation with ϱ.

$$\forall n\colon\ n \overset{\mathbf{Of_c}}{\rightsquigarrow} \varrho \implies n \doteq \mathtt{[]}$$

The notation $n \doteq expr$ means that the node n of the abstract syntax tree is the root node of the subtree representing the Erlang expression *expr*.

(b) If ϱ is on a recursive path, then ϱ must evaluate to a non-empty list, i.e. in the compact data-flow reaching only expressions in the syntactic form `[`χ`|`τ`]` or `[`χ`]` `++` τ (where χ and τ are arbitrary expressions).

$$\begin{aligned}\forall n\colon\ & n \overset{\mathbf{Of_c}}{\rightsquigarrow} \varrho \implies \\ & \big[\big(\exists \chi, \tau\colon n \doteq \mathtt{[}\chi\mathtt{|}\tau\mathtt{]}\big) \vee \\ & \ \big(\exists \sigma, \tau\colon n \doteq \sigma\mathtt{++}\tau \wedge \\ & \qquad \big(\forall n'\colon n' \overset{\mathbf{Of_c}}{\rightsquigarrow} \sigma \implies \exists \chi\colon n' \doteq \mathtt{[}\chi\mathtt{]}\big)\big)\big]\end{aligned}$$

With respect to the above condition, we set out further requirements on all possible χ and τ.

(i) χ depends on `L` only through Head`L`, namely: for each $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} \chi$ path in the dependence graph $G_D f$,

there exists $n \in \text{Head}\mathtt{L}$ such that the path is a concatenation of paths $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} n$ and $n \overset{\mathbf{dep}}{\rightsquigarrow} \chi$.

(ii) τ is the result of the recursive call of f, where the parameters of the call are $\overline{V_1}$, the tail of $\mathtt{L}$, and $\overline{V_2}$. More precisely, $\exists n\ \exists\overline{\nu_1}\ \exists\alpha\ \exists\overline{\nu_2}\ \exists n'$:

$$n \doteq f\overline{\nu_1}, \alpha, \overline{\nu_2} \ \wedge\ \overline{V_i} \overset{\mathbf{Of}}{\rightsquigarrow} \overline{\nu_i}\ i = 1,2 \ \wedge$$
$$n' \in \text{Tail}\mathtt{L} \ \wedge\ n' \overset{\mathbf{Of}}{\rightsquigarrow} \alpha \ \wedge\ n \overset{\mathbf{Of}}{\rightsquigarrow} \tau.$$

5.4.2 Tail-recursive map-like functions. For efficiency reasons, Erlang programmers often write recursive functions in tail-recursive style. Map-like functions can also be implemented like that. There are many syntactic forms a tail-recursive map-like function can take. (Remember also that mutually recursive functions can also be considered map-like.) One possible syntactic appearance of a tail-recursive map-like function is the following.

```
f(Parameter, Acc, List) ->
  case List of
    [] -> Acc;
    [Head|Tail] ->
      X = ... Head ...,
      NewAcc = [X|Acc],
      f(Parameter,NewAcc,Tail)
  end.
```

In our approach a function is recognized as a map-like tail-recursive function if the following conditions hold. The function should have two list parameters (an input list and an accumulator), and return the accumulator in its base case. The accumulator is extended during the recursive calls; an item added to the accumulator may depend on the head of the input list, but may not depend on its tail. In the recursive case, the last expression to evaluate is a recursive call on the tail of the input list and the new accumulator. The function may have additional parameters, but all these parameters must be passed to the recursive call unchanged.

To formally define the rules, we consider a function f, as well as the control-flow graph $G_{CF}f$ and dependence graph G_Df built from f as a starting point. Some of the rules coincide with the ones presented in the previous section.

(1) f must be recursive.
(2) The definition of f must have a base case.
(3) f should not be of divide-and-conquer style.
(4) f may have multiple parameters, among which there are two lists: one to be processed element-by-element, and an accumulator to store the result of the computation. For simplicity, we may again assume that the definition of f is provided with a single clause, and its head is in the form of $f\overline{V}, \mathtt{Acc}, \mathtt{L}$, where parameter $\mathtt{L}$ corresponds to the list processed element-wise by f, and $\mathtt{Acc}$ is the accumulator. In the following the already defined semantic functions HeadL and TailL, as well as the set R of return expressions are also used.
(5) For each return expression $\varrho \in R$, the following requirements must hold.

(a) If ϱ is on a non-recursive execution path, then ϱ must evaluate to the same value as accumulator. Namely, if node n represents the formal parameter `Acc`, then

$$n \overset{\mathbf{Of_c}}{\rightsquigarrow} \varrho.$$

(b) If ϱ is on a recursive path, then ϱ must evaluate to an appropriate recursive call of f, i.e. for all nodes n such that $n \overset{\mathbf{Of_c}}{\rightsquigarrow} \varrho$ there exist nodes $\overline{\nu}, \alpha, \beta$ such that $n \doteq f\overline{\nu}, \alpha, \beta \ \wedge\ \overline{V} \overset{\mathbf{Of}}{\rightsquigarrow} \overline{\nu} \ \wedge\ \beta \in \text{Tail}\mathtt{L}$, moreover, for all nodes n' such that $n' \overset{\mathbf{Of_c}}{\rightsquigarrow} \alpha$, either $n' \doteq [\chi|\tau]$ for some χ and τ, or there exist σ and τ such that $n' \doteq \sigma{+}{+}\tau$ and

$$\forall n'' \colon n'' \overset{\mathbf{Of_c}}{\rightsquigarrow} \sigma \implies \big(\exists\chi \colon n'' \doteq [\chi]\big).$$

With respect to the above condition, we set out further requirements on all possible χ and τ.

(i) χ depends on $\mathtt{L}$ only through HeadL, namely: for each $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} \chi$ path in the dependence graph G_Df, there exists $n \in \text{Head}\mathtt{L}$ such that the path is a concatenation of paths $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} n$ and $n \overset{\mathbf{dep}}{\rightsquigarrow} \chi$.
(ii) τ must evaluate to the value of the accumulator:

$$\forall n\colon \quad n \overset{\mathbf{Of_c}}{\rightsquigarrow} \tau \implies n \doteq \mathtt{Acc}.$$

5.4.3 Weakening the conditions. In certain cases the restriction on the parameters of the recursive calls can be relaxed. In the previous section rule 5b ensured that the parameters other than the accumulator and the list processed element-wise should be the same for all recursive calls ($\overline{V} \overset{\mathbf{Of}}{\rightsquigarrow} \overline{\nu}$). The following example illustrates a feasible generalization to this approach.

```
g(List)->
 f(1, [], List).

f(N, Acc, List) ->
  case List of
    [] -> Acc;
    [Head|Tail] ->
      X = someHeavyComputation(Head,N),
      f(N+1,[X|Acc],Tail)
  end.
```

This code can be refactored in such a way that the computation is split into two phases: in the first phase the input is prepared for the second phase, and the second phase becomes a map-like function according to the rules of the previous section.

```
g(List)->
  f([], lists:zip(List, generate(1,List))).

generate(N, List) ->
  case List of
    [] -> [];
    [Head|Tail] ->
      [N | generate(N+1,Tail)]
```

```
  end.

f(Acc, List) ->
  case List of
    [] -> Acc;
    [{Head, N}|Tail] ->
      X = someHeavyComputation(Head,N),
      f([X|Acc],Tail)
  end.
```

5.4.4 Foreach-like functions. The behaviour of a *foreach-like function* is similar to a map-like function: it applies the same operation on each element of the processed list. The difference between the two is that the return value of a foreach-like function is ignored: it is executed only for the sake of its side-effects. Usually the `ok` atom is used as return value in foreach-like functions. The computational structure of such functions is shown in the following definition (of course many other syntactic forms can be used to describe this behaviour).

```
f(Parameter,List) ->
  case List of
    [] -> ok;
    [Head|Tail] -> ... Head ...,
                   f(Parameter,Tail)
  end.
```

It is possible to refactor this code into a task farm using PaRTE, although the tool follows a quite generic rewrite scheme, and hence the result of the rewriting seems overly complicated.

```
f(Parameter,List) ->
  Fun = fun(H) ->
          case [H | ok] of
            [] -> ok;
            [H | ok] -> ... H ...,
                        ok
          end
        end,
  skel:do([{farm, [{seq, Fun}], 10}], List),
  ok.
```

In this particular situation the `case`-expression could be eliminated with a clean-up refactoring (not yet implemented in PaRTE), so one can arrive at the following definition.

```
f(Parameter,List) ->
  Fun = fun(H) ->
            ... H ...,
            ok
        end,
  skel:do([{farm, [{seq, Fun}], 10}], List),
  ok.
```

The rules for identifying foreach-like functions are:

(1–4) f must be recursive, must have a base case, and should not be divide-and-conquer style. As before, we assume here that the definition of f is provided with a single clause, the head of which is $f\overline{V_1}, \mathtt{L}, \overline{V_2}$, where parameter $\mathtt{L}$ corresponds to the list consumed element-by-element by f.

(5) Every expression on every execution path of function f (except the expressions that has effect on the recursive call) must not depend on $\mathtt{L}$ through Tail$\mathtt{L}$.

$$\forall e : \left(\exists n \in \mathrm{Tail}\mathtt{L} : \mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} n \ \wedge n \overset{\mathbf{dep}}{\rightsquigarrow} e\right) \implies \left(\exists c : e \overset{\mathbf{dep}}{\rightsquigarrow} call_f^c\right)$$

(6) No recursive call of f may depend on $\mathtt{L}$ through Head$\mathtt{L}$: for all $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} call_f^c$ path in the dependence graph $G_D f$, $\nexists n \in \mathrm{Head}\mathtt{L}$ such that the path is a concatenation of paths $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} n$ and $n \overset{\mathbf{dep}}{\rightsquigarrow} call_f^c$.

(7) The definition of f passes the tail of the input list to each recursive call, together with all other received parameters, i.e. for each $f\overline{\nu_1}, \alpha, \overline{\nu_2}$ recursive call:

$$\overline{V_i} \overset{\mathbf{0f}}{\rightsquigarrow} \overline{\nu_i} \ i = 1,2 \ \wedge \ \forall n : n \overset{\mathbf{0f}}{\rightsquigarrow} \alpha \implies n \in \mathrm{Tail}\mathtt{L}.$$

5.5 Stream-based computation

Stream-based recursive functions iterate some computation on *received* data. In this section we characterize map-like and foreach-like stream-based computations.

5.5.1 Map-like stream processing. Map-like functions may iterate over the elements of a list, or – which is quite common in Erlang programs – over a stream: values read from the message box of a process. Skel has a general interface to provide the inputs of the task farm skeleton as a callback module. For this reason, PaRTE can identify stream processing code fragments as pattern candidates, and provide automated transformations to prepare the appropriate callback module for Skel. We only give here a sketch of how such an analysis and transformation can be carried out.

In pattern candidate discovery, we search for functions satisfying the following properties. The function must be tail-recursive. It should operate on a state of type `list()`. It should contain a single `receive` expression in its body. In the recursive execution path, we process the received data, and prepend the result to the state. The operation on the received data must not depend on the state. (The state is only used as an accumulator.) As an example, consider the following function definition.

```
computation() ->
  loop([]).

loop(State) ->
  receive
    extr -> State;
    X    -> Val = g(X+1),
            loop([Val | State])
  end.
```

In order to introduce Skel-based parallelism, a callback module has to be provided, which contains a `next_input/1` function producing the elements of the input stream.

```
computation() ->
  loop().

loop() ->
  skel:do([ {farm,
             [{seq, fun(X) -> g(X+1) end}],
             10} ],  % 10 worker tasks
           ?MODULE).  % name of callback module

next_input(ok) ->
  receive
    extr -> {eos,ok};
    X    -> {input,X,ok}
  end.
```

The formalization of the identification rules uses the control-flow graph $G_{CF}f$ and dependence graph G_Df built starting from function f. The set of result expressions in f will be denoted by R in the rules.

(1–3) f must be recursive, must have a base case, and should not be divide-and-conquer style.

(4) The function must be tail recursive:

$$\forall p \in EPstart_f \text{ and } \forall c_1 \text{ such that } call_f^{c_1} \in p\text{:}$$

$$\forall q \in EPret_f^{c_1}\ \forall n \in q\text{:}\ \ n = end_f \vee \exists g \exists c_2\text{: } n = ret_g^{c_2}.$$

(5) f may have multiple parameters, among which there is one, a list, which will act as the state, and be *extended* during the recursive calls. We assume that the definition of f is provided with a single clause, the head of which is $f\overline{V_1}, \mathtt{L}, \overline{V_2}$, where parameter `L` corresponds to the state.

(6) For each return expression $\varrho \in R$ the following requirements must hold.

(a) ϱ is the last expression of a branch β in a **`receive`** expression.

$$\forall n : \varrho \xrightarrow{\textbf{flow}} n \implies \left(\exists \overline{\beta} : n \doteq \texttt{receive}\ \overline{\beta}\ \texttt{end}\right)$$

(b) If ϱ is on a non-recursive execution path, then ϱ must evaluate to the state, i.e. `L`:

$$\mathtt{L} \overset{\textbf{Of}}{\rightsquigarrow} \varrho$$

(c) If ϱ is on a recursive execution path, then ϱ must be a recursive call to f:

$$\varrho \doteq f\overline{\nu_1}, \nu, \overline{\nu_2}$$

for some actual parameters $\overline{\nu_1}$, ν and $\overline{\nu_2}$, moreover these parameters are $\overline{V_1}$, the extended state and $\overline{V_2}$, namely $\overline{V_i} \overset{\textbf{Of}}{\rightsquigarrow} \overline{\nu_i}$ $i = 1, 2$ and

$$\begin{aligned}\forall n\text{:}\ \ & n \overset{\textbf{Of}_\textbf{c}}{\rightsquigarrow} \nu \implies \\ & \big[\left(\exists \chi, \tau\text{: } n \doteq [\chi \,|\, \tau]\right) \vee \\ & \ \left(\exists \sigma, \tau\text{: } n \doteq \sigma \texttt{++} \tau \wedge \right. \\ & \qquad \left.\left(\forall n'\text{: } n' \overset{\textbf{Of}_\textbf{c}}{\rightsquigarrow} \sigma \implies \exists \chi\text{: } n' \doteq [\chi]\right)\right)\big].\end{aligned}$$

With respect to the above condition, we set out further requirements on all possible χ and τ. Let α denote the received expression, and φ denote the operation performed on this expression.

(i) χ is the result of the operation performed on the received data.

$$\varphi\alpha \overset{\textbf{Of}}{\rightsquigarrow} \chi$$

(ii) τ is the original value of the state.

$$\mathtt{L} \overset{\textbf{Of}}{\rightsquigarrow} \tau$$

(iii) The operation on the received data must not depend on the state:

$$\neg\left(\mathtt{L} \overset{\textbf{dep}}{\rightsquigarrow} \varphi\alpha\right)$$

5.5.2 Foreach-like stream processing. In contrast to map-like stream processing, foreach-like stream processing does not accumulate computed values in a state: for each incoming message a computation with some side-effect is performed, the result of which is discarded. The following code fragment exemplifies foreach-like stream processing.

```
loop() ->
  receive
    extr -> ok;
    X    -> computation(X),
            loop()
  end.
```

Both the discovery and the transformation of such definitions are easier than those of map-like stream processing. Refactoring the `loop/0`function to a task farm with 10 worker processes, and providing a callback module for reading the stream content results in the following code.

```
loop() ->
  skel:do([ {farm,
             [{seq, fun(X)-> computation(X) end}],
             10} ],
           ?MODULE).

next_input(ok) ->
  receive
    extr -> {eos,ok};
    X    -> {input,X,ok}
  end.
```

The rules for the identification of a foreach-like function f working on streams are the following. As before, we assume that f is defined with a single clause, the head of which is in the form $f\overline{V}$.

(1–4) f must be recursive, must have a base case, and should not be divide-and-conquer style; furthermore, it should be tail-recursive.

(5) For each return expression $\varrho \in R$ the following requirements must hold.

(a) ϱ is the last expression of a branch β in a **`receive`** expression.

$$\forall n : \varrho \xrightarrow{\textbf{flow}} n \implies \left(\exists \overline{\beta} : n \doteq \texttt{receive}\ \overline{\beta}\ \texttt{end}\right)$$

(b) In every recursive execution path, the parameter of the tail recursive call is $\overline{V}$. Formally, if $\varrho \doteq f\overline{\nu}$ for some actual parameters $\overline{\nu}$, then

$$\overline{V} \overset{\mathbf{0f}}{\rightsquigarrow} \overline{\nu}.$$

5.6 Pipeline candidates

Pipelines consist of multiple computational stages, and perform well if a large number of independent data items are provided as input. Each data item is individually passed through the pipeline, where each stage transforms it. Thus, a pipeline is also an iterative model that applies the same computations (the stages of the pipeline) to every data item. In terms of pattern candidate discovery, pipelines are similar to task farms with an additional requirement on the computation applied on each data item. Technically, whenever a task farm candidate is found, one can check whether it can also be considered as a pipeline candidate. The following additional requirement applies to the χ expression in the rules of the list- and stream-processing map- and foreach-like functions.

- There exist at least two functions g and h representing some (preferably complex) computations, as well as c_1 and c_2 call sites such that:

$$\exists p \in EPstart_f : \ \chi \in p \wedge call_g^{c_1} \in p \wedge call_h^{c_2} \in p.$$

5.7 Reduce pattern candidates

The *reduce pattern* models a data parallel computation where the input items (e.g. a list of integers) are processed by an associative (and possibly commutative) operation (e.g. integer addition). The Skel-library contains a reduce skeleton to implement this useful pattern.

Possible pattern candidates include certain functions in the standard library (see Section 5.3) and functions following a certain recursion scheme. The latter can be illustrated by the following code example. As we can see, the head of the input list and the result of the recursive call on the tail of the input list are combined using the associative (and possibly commutative) operation.

```
f(L) ->
  case L of
    [ Head | Tail] ->
      V = f(Tail),
      some_operation(Head, V);
    [] -> default_value
  end.
```

Such a definition can be refactored to a parallel implementation based on Skel. (We may also need some clean-up transformations, such as dead code elimination and variable elimination).

```
f(L) ->
  F = fun(X, Y) -> some_operation(X, Y) end,
  skel:do([{reduce, F, fun(X)-> X end}],L).
```

In order to formalize the candidate identification rules, we follow our recipe. A function f is investigated, and we assume that it is defined with a single clause, and its head is in the form $f\overline{V_1}, \mathtt{L}, \overline{V_2}$, where `L` is the list parameter holding the items to be reduced into a single value.

(1–3) f must be recursive, must have a base case, and it should not be of divide-and-conquer style.

(4) For each return expression $\varrho \in R$, the following requirements must hold.

(a) If ϱ is on a recursive path, then it must come from the application of an associative binary operation (function or operator) φ.

$$\forall n: \ n \overset{\mathbf{0f_c}}{\rightsquigarrow} \varrho \implies \left(\exists \alpha_1, \alpha_2: \ n \doteq \varphi(\alpha_1, \alpha_2)\right)$$

$\forall \alpha_1, \alpha_2, \alpha_3 : \varphi\varphi\alpha_1, \alpha_2, \alpha_3 = \varphi\alpha_1, \varphi\alpha_2, \alpha_3$
With respect to the above condition, we set out further requirements on all possible α_1 and α_2.

(i) α_1 depends on `L` only through Head`L`, namely: for each $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} \alpha_1$ path in the dependence graph, there exists $n \in$ Head`L` such that the path is a concatenation of paths $\mathtt{L} \overset{\mathbf{dep}}{\rightsquigarrow} n$ and $n \overset{\mathbf{dep}}{\rightsquigarrow} \alpha_1$.

(ii) α_2 is the result of the recursive call of f, where the parameters of the call are $\overline{V_1}$, the tail of `L`, and $\overline{V_2}$. Formally,

$$\forall n : n \overset{\mathbf{0f}}{\rightsquigarrow} \alpha_2 \implies \left(\exists \overline{\nu_1}, \alpha, \overline{\nu_2} : n \doteq f\overline{\nu_1}, \alpha, \overline{\nu_2} \ \wedge \ \overline{\nu_i} \ i = 1, 2 \ \wedge \left(\forall n' : \ n' \overset{\mathbf{0f}}{\rightsquigarrow} \alpha \implies n' \in \mathrm{Tail}\mathtt{L}\right)\right)$$

(b) The implementation of the reduce skeleton in Skel assumes that φ is also commutative. Therefore, if we want to introduce this skeleton, the following is also required.

$$\forall \alpha_1, \alpha_2 : \ \varphi\alpha_1, \alpha_2 = \varphi\alpha_2, \alpha_1$$

5.8 Divide&conquer candidates

The divide and conquer (*d&c*) pattern describes a computation where a problem is recursively *divided* into sub-problems (until a given termination condition is established), and after solving the sub-problems, the solutions are recursively *combined* to produce the solution of the overall problem. There are many syntactic forms that express this behaviour. An example was presented in Section 3.4, and now two more are shown here.

```
f(L) ->
  case isbase(L) of
    true -> base(L);
    false ->
      SubLists = divide(L),
      SubSols  = lists:map(fun f/1,SubLists),
      combine(SubSols)
  end.

qs(List) ->
  case List of
    [] -> [];
    [Head | Tail] ->
```

```
    {L1, L2} = lists:partition(
                          fun(X) -> X < Head end,
                          Tail ),
    SortedL1 = qs(L1),    % items <  Head
    SortedL2 = qs(L2),    % items >= Head
    SortedL1 ++ [Head] ++ SortedL2
end.
```

The first example here illustrates the canonical *d&c* structure, which can directly be transformed into a parallel pattern with PaRTE. The second example is an implementation of the well-known "functional quick-sort algorithm".

We can observe that a function is a divide-and-conquer candidate if

- it has a list parameter,
- it has multiple recursive calls in its body,
- the parameters of these calls do not depend on each other,
- these parameters are originating from the list parameter of the function, and
- the return value of the function depends on the result of the recursive calls.

Here we focus on list-processing *d&c* candidates – a more generic *d&c* discovery is defined by Kozsik *et al.* (2016) [26]. Moreover, the technique of step-wise transformation of a *d&c* candidate into a parallel pattern is presented by Kozsik *et al.* (2017) [27].

The formal identification rules for a *d&c* candidate are the following. As usual, we consider a function f, and for simplicity we assume that its definition is given in one clause, with a head in the form $f\overline{V_1}, \mathtt{L}, \overline{V_2}$, where parameter `L` corresponds to a list.

(1–2) f must be recursive and must have a base case.

(3) f must have multiple recursive calls in its body, as characterized by the following three alternative possibilities.

- f may have an execution path that contains at least two independent recursive calls:

$$\exists c_1, c_2, \exists p \in EPret_f^{c_1} \text{ such that}$$

$$call_f^{c_2} \in p \ \wedge \ \forall a \in \mathrm{ARG} c_2 : \ \neg\left(a \overset{\mathbf{dep}}{\rightsquigarrow} ret_f^{c_1}\right)$$

  where ARG is the set of nodes representing the arguments of a function call.
- f may have an execution path containing a list comprehension with head expression h, which calls f directly or indirectly.

$$\exists p \in EPh, \ \exists c \text{ such that } call_f^c \in p$$

- f may (directly or indirectly) call a *map* or a map-like function g, which in turn calls f in its every recursive execution path.

$$\exists p \in EPstart_f, \exists c_1, \exists g \text{ map-like function}$$

such that

$$call_g^{c_1} \in p \ \wedge \ \forall q \in EPstart_g :$$

$$\left(\exists c_2 : \ call_g^{c_2} \in q\right) \implies \left(\exists c_3 : \ call_f^{c_3} \in q\right)$$

(4) For each return expression $\varrho \in R$, where ϱ is on a recursive execution path, the following requirements must hold.

(a) ϱ must be the result of the combine function/expression γ, where

(b) γ may depend on the $\overline{V_1}$ and $\overline{V_2}$ parameters of f, and also depends on the result of the recursive function calls, and

(c) each parameter of the recursive function calls must be a sub-list or a transformed sub-list of `L` produced by the divide function δ.

5.8.1 Sort and search patterns. The *sort pattern* takes an input list as an argument, and produces a sorted output. The *search pattern* looks for items in a list which satisfy a given predicate. Both patterns can be implemented in a *d&c* style, therefore we can identify candidates of these pattern with the *d&c* discovery rules.

5.9 Pool candidates

The *pool pattern* is a computation working on a set of values (a "pool"); it applies an "evolution function" on some elements of the pool, and the result of such an application may be (selectively) pushed back into the pool. This is repeated until a given "termination condition" is established [49].

A generic syntactic scheme for an Erlang code fragment which implements a pool behaviour is the following.

```
f(Pool) ->
  case terminated(Pool) of
   true  -> Pool;
   false ->
     Selected = ...Pool...,
     Evolved  = lists:map(fun evolve/1, Selected),
     Filtered = ...Evolved...,
     NewPool  = ...Pool...Selected...Filtered...
     f(NewPool)
end.
```

A candidate for a pool is a function f satisfying the following conditions.

(1) f must be tail recursive and must have a base case.

(2) f may have multiple parameters (we assume that the head of its definition is in the form of $f\overline{V_1}, \mathtt{Pool}, \overline{V_2}$), among which there is one, a list `Pool`, representing the pool.

(3) f has an expression in its body that expresses the iterative evaluation of the "evolution" function:
- a list comprehension,
- a `lists:map/2` call,
- a call to a map-like recursive function.

(4) The input to the above iteration originates from parameter `Pool`. It may be `Pool`, or a sublist, or σ`Pool`, where σ is the selection function.

(5) The output of the iteration is possibly filtered by a function φ.

(6) The return expression $\varrho \in R$ of the function f is:
- `Pool` in the base case, or

- a recursive call $f\overline{\nu_1}$, `NewPool`, $\overline{\nu_2}$ in the recursive branches, where parameters $\overline{V_i}$ flow into $\overline{\nu_i}$ $i = 1, 2$ unchanged, and `NewPool` depends on $\overline{V_i}$, on the selected elements produced by σ, and on the filtered elements produced by φ.

5.9.1 Orbit pattern. The *orbit pattern* models an iterative construction of a set by a few generator functions until the transitive closure of an initial set is reached. The orbit pattern can be implemented in terms of the pool pattern, therefore the general pool identification algorithm can recognise it as a pool pattern.

6 EVALUATION

Pattern discovery identifies source code fragments that are amenable for parallelization, in particular those fragments that can be transformed into any of the available algorithmic skeletons. This section presents statistics on the different kinds of pattern candidates found in some analysed projects. The tables below show the number of occurrences of certain syntactic forms that satisfy the requirements put on the *task farm*, *reduce* or *divide-and-conquer* patterns. Note that *pipeline* candidates are also task farm candidates. The tables show the number of the latter, while the figure for the former can be found as a comment below each table.

We verified manually (some of) the pattern candidates found by the tool, to see whether they really possess the properties that qualify them for the identified pattern. Most candidates (list comprehensions and applications of predefined higher-order functions) are trivially appropriate, hence we checked them only by random sampling. Map-like functions and divide-and-conquer algorithms are more sophisticated to characterize and harder to find, and so we paid more attention to them. We checked the dozen map-like functions the tool found, but used only random sampling for the about one hundred divide-and-conquer candidates. Some interesting examples are presented later as well.

Ant Colony Optimization for the Single Machine Total Weighted Tardiness Problem. This example application has been developed in the ParaPhrase project, and its source code can be found in the project website [44].

The analyzed code contains 56 function definitions in 21 files, and consists of 483 effective lines of code.

Candidate	Number of occurrences	Kind of pattern
various library calls	10	farm

Table 1: Ant Colony Optimization

Intensional Computing Engine. This application is an evaluator of the abstract syntax tree for the ICE language, which is an implementation of intensional programming.

The source code for this framework can be found here:

https://github.com/esl/ice

The analyzed code contains 141 function definitions in 21 files, and consists of 1094 effective lines of code.

Candidate	Number of occurrences	Kind of pattern
various library calls	7	farm
various library calls	4	reduce

Table 2: Intensional Computing Engine

Image merging. Different implementations of a simple image processing algorithm. The source code can be found in the ParaPhrase project website [44].

Candidate	Number of occurrences	Kind of pattern
various library calls	19	farm
various library calls	2	reduce
list comprehension	10	farm
map-like function	5	farm

Table 3: Image merging

The analysed code contains 104 function definitions in 6 files, and consists of 779 effective lines of code.

Out of the 34 farm candidates, 4 occurrences were pipeline candidates as well.

MAS – a framework for multi-agent systems. The source code for this framework supporting the creation of multi-agent systems can be found here.

https://github.com/ParaPhraseAGH/erlang

The analysed code contains 177 function definitions in 25 files, and consists of 1646 effective lines of code.

Candidate	Number of occurrences	Kind of pattern
various library calls	7	farm
various library calls	16	reduce
list comprehension	63	farm
map-like function	1	farm

Table 4: MAS framework

Out of the 71 farm candidates, 20 occurrences were pipeline candidates as well.

Thorn – a map-reduce framework. The source code for this project can be downloaded from here.

https://github.com/ICGog/MapReduce-Thorn

The analyzed code contains 158 function definitions in 15 files, and consists of 1313 effective lines of code.

Candidate	Number of occurrences	Kind of pattern
various library calls	17	farm
various library calls	5	reduce

Table 5: Thorn framework

Mnesia – a distributed database management system. Mnesia is part of the standard Erlang/OTP system, and its source code is available here.

https://github.com/erlang/otp/tree/maint/lib/mnesia/src

The analyzed code contains 1693 function definitions in 31 files, and consists of 22653 effective lines of code.

Candidate	Number of occurrences	Kind of pattern
various library calls	72	farm
various library calls	36	reduce
list comprehension	58	farm
map-like function	5	farm
d&c-like function	57	divide and conquer

Table 6: Mnesia

From the farm candidates we have identified 8 as a pipe candidate.

RefactorErl – a source code analysis and transformation tool. We have analyzed some components of the RefactorErl tool. Its source code is available here.

http://plc.inf.elte.hu/erlang/dl/refactorerl-0.9.14.09.zip

The analyzed `referl_core` component contains 1534 function definitions in 53 files, and consists of 19694 effective lines of code.

Candidate	Number of occurrences	Kind of pattern
various library calls	139	farm
various library calls	55	reduce
list comprehension	347	farm
map-like function	3	farm
d&c-like function	31	divide and conquer

Table 7: RefactorErl: referl_core

From the farm candidates we have identified 49 as pipe candidate.

6.1 Some interesting candidates

During the validation of the pattern candidate identification technique, we have encountered really nice instances of map-like and divide-and-conquer definitions. Here we point out four demonstrative cases.

The first example (Figure 12) is the `mnesia_tm:reverse/1` function from the code of the Mnesia database-management system. This shows that real-world code may contain trivial re-implementations of `lists:map/2` (or the equivalent list comprehension). The identification of such redundancies can be useful not only from the point of view of parallelization, but also in general.

```
reverse([]) ->
  [];
reverse([H=#commit{ram_copies=Ram, disc_copies=DC,
           disc_only_copies=DOC,snmp = Snmp}
         |R]) ->
  [
   H#commit{
     ram_copies       =  lists:reverse(Ram),
     disc_copies      =  lists:reverse(DC),
     disc_only_copies =  lists:reverse(DOC),
     snmp             = lists:reverse(Snmp)
    }
   | reverse(R)].
```

Figure 12: Mnesia map-like function

A slightly more complex, but still map-like definition is coming from the MAS framework (Figure 13), the function `misc_utils:count_funstats/2`. Again, it is also very easy to turn this into a list comprehension, but for some reason the programmer decided to code it with explicit recursion.

```
count_funstats(_,[]) ->
 [];
count_funstats(Agents,
               [{Stat, MapFun,
                 ReduceFun, OldAcc}|T]) ->
  NewAcc =
    lists:foldl(ReduceFun,
                OldAcc,
                [MapFun(Agent)
                  || Agent <- Agents]),
  [{Stat, MapFun, ReduceFun, NewAcc}
    | count_funstats(Agents,T)].
```

Figure 13: MAS map-like function

The third example (Figure 14) shows a beautiful instance of the divide-and-conquer pattern: the function `refcore_callanal:listcons_length/2` almost completely follows the "canonical form" of *d&c*. It operates on a list; it splits the list in the divide-phase using `lists:partition/2`, it applies itself recursively with `lists:map/2` in the non-base case, and finally combines the results explicitly (with `lists:append/1`). Note that the combination of `lists:append/1` and `lists:map/2` could have been expressed with `lists:flatmap/2` – which could make the automatic

transformation of this pattern harder (or at least it would require built-in knowledge about `lists:flatmap/2`).

```
...
listcons_length(N, #expr{}) ->
  Ns = ?Dataflow:?reach([N], [back], true),
  L1 = [N2 || N2 <- Ns, N2 /= N,
              ?Graph:class(N2) == expr],
  {L2, L3} =
    lists:partition(fun is_cons_expr/1, L1),
  if L2 == [] orelse L3 /= [] ->
       incalculable;
     true  ->
       lists:append(
         lists:map(fun listcons_length/1,
                   L2))
  end;
...
```

Figure 14: Nice *D&C* in RefactorErl

The fourth example is again from the code of RefactorErl, namely the function `refcore_pp:realtoken_neighbour/3` (Figure 15). It is clearly a divide-and-conquer definition, because it calls `realtoken_neighbour_/4`, which iterates through the list `Parents`, and calls `realtoken_neighbour/3` in each step (a `map` is encoded in its body). This definition is really hard to transform to a divide-and-conquer skeleton.

7 RELATED WORK

Various approaches have been proposed related to the identification and semi-automatic transformation of source code fragments that are amenable to parallelisation. Some of these methods use purely static information, such as a compile-time optimisation tool that produces parallel code. Others are monitoring the dynamic behaviour of the system as well.

The approach proposed by Hammacher *et al.* [22] uses the latter method. Their tool examines execution traces based on JavaSlicer, and recommends the independent program paths in Java code to run in parallel. It uses a dynamic dependence graph to calculate the paths. It does not provide automatic transformations to turn the source code fragments to instances of parallel patterns. The user can introduce parallelism manually. Our parallel pattern candidate discovery is based on static analysis, but we could also integrate a static analysis based approach into one using run-time monitoring of executions – however, this is left as future work.

Similar to our approach, the techniques developed by Shane *et al.* [33, 34] and Wloka *et al.* [48] use static analyses to introduce parallelism. The goal of these researches is the parallelisation of X10 and Java programs by refactoring, the discovery of parallelism bottlenecks and the insurance of thread safety and reentrancy. Compared to our approach, they neither consider patterns as instances of algorithmic skeletons, nor provide suggestions on which skeletons to apply.

The tool developed by Molitorisz *et al.* [37, 38] is able to discover some rather basic parallelization patterns, and proposes supporting refactoring as well. The tool performs static analysis on Java programs to determine data independent program parts, and inserts parallel constructs directing asynchronously parallel execution. This tool neither works with algorithmic skeletons, nor presents the pattern candidates to the user.

There are further useful tools for Java. Dig *et al.*, for instance, introduce refactorings that are able to turn sequential Java code to concurrent using generic Java Concurrency libraries to control parallel execution [16–18].

For C, Mak *et al.* [32] describe a set of automatic shaping and parallelization based on dependence profiling. Freisleben and Kielmann introduces an automated transformation framework was to transform sequential *d&c* algorithms, written in C, to parallel equivalents [21]. The transformations use a few annotations (which are required to be provided by the programmer) to identify the places where parallelism should be introduced. Our tool is not able to directly introduce *d&c* skeletons, but it can automatically identify *d&c* candidates [26], and provides small refactorings, which allow the user to introduce *d&c* step-by-step [27].

Within the functional language research community, Aronis *et al.* proposed an ad hoc parallelization approach using Dialyzer (the discrepancy analyser for Erlang) and a suite of Erlang benchmarks [3, 4]. They did not apply structured parallelism or parallel skeletons.

Several skeleton libraries exist for a number of mainstream programming languages, such as Java, C or C++. The skeleton research community has been working on methods for parallel programming in high-level languages since the nineties [13–15]. There are research activities that focus on refactoring support for skeleton introduction as well; for example, Brown, Janjic *et al.* [10] proposed a few basic refactorings to introduce task farm and pipeline skeletons into C++ using the FastFlow library. PaRTE is unique in the sense that it integrates pattern discovery, candidate ranking, and refactorings in a single tool.

Brown *et al.* (2011) introduced a limited number of parallel refactorings [11] in *HaRe* – the Haskell Refactorer. They introduce parallelism using structural refactorings, but they do not provide support for pattern discovery.

The algorithmic skeletons targeted by our pattern discovery process are implemented in the Skel algorithmic skeleton library for Erlang [43]. An extension to Skel [50] defines some high-level pattern implementations, including *d&c*. Two useful features of this library are that we can limit the number of started parallel processes, and force sequential evaluation where beneficial. Although Skel does not support distributed skeletons such as Eden [31] or D-Clean [51], it is possible to extend it based on the Erlang concepts.

Bozó *et al.* (2014) introduces the initial concepts of the ParaPhrase Refactoring Tool for Erlang [7]. Besides the pattern discovery and the semi-automatic pattern introduction by refactorings, PaRTE provides candidate ranking based on performance measurements and estimates as well. The cost

```
realtoken_neighbour(Node, DirFun, DownFun) ->
  case lists:member(?Graph:class(Node), [clause,expr,form,typexp,lex]) of
    false -> no;
    _ ->
      case ?Syn:parent(Node) of
        [] -> no;
        [{_,Parent}] ->
          case lists:dropwhile(fun({_T,N}) -> N/=Node end, DirFun(?Syn:children(Parent))) of
            [{_,Node},{_,NextNode}|_] ->  DownFun(NextNode);
            _ -> realtoken_neighbour(Parent, DirFun, DownFun)
          end;
        Parents -> realtoken_neighbour_(Parents, DownFun(Node), DirFun, DownFun)
    end
  end.

%Implementation helper function for realtoken_neighbour/3
realtoken_neighbour_([], _FirstLeaf,_DirFun,_DownFun) ->
  no;
realtoken_neighbour_([{_,Parent}|Parents], FirstLeaf, DirFun, DownFun) ->
  case realtoken_neighbour(Parent, DirFun, DownFun) of
    FirstLeaf -> realtoken_neighbour_(Parents, FirstLeaf, DirFun, DownFun);
    NextLeaf  -> NextLeaf
  end.
```

Figure 15: RefactorErl complex *D&C*

models used by the tool were defined by Brown, Danelutto *et al.* [9].

In a former publication [6] we already provided the formal definition of "map-like functions" in order to automatically discover list-based element-wise computations. Moreover, we described a variety of program shaping transformations to refactor the map-like pattern candidates in a syntactic form that can be then transformed into an application of the task farm skeleton. The refactorings were defined formally as well by Horpácsi *et al.* [25]. A refactoring language was also designed to specify the transformations in a verifiable way.

Optimizing compilers are able to identify parallelizable code fragments, and also to parallelize them automatically. For example, SkelML [36] is a parallel skeleton-based compiler for SML. It can automatically identify applications of certain higher-order functions as pattern candidates, and transform them to applications of equivalent parallel skeletons. SkelML does not present pattern candidates to the programmer or guide the programmer in the parallelisation activity. The pattern discovery technique applied by SkelML is less generic than ours, since the strength of our tool is the analysis of user defined recursive function calls.

8 SUMMARY

The ParaPhrase Refactoring Tool for Erlang facilitates the parallelization of Erlang source code by providing automated pattern candidate discovery and refactoring transformations. This paper presents the underlying methodology, and teaches the use of the tool. Moreover, we explain how the static source code analyses work for parallel pattern candidate discovery – this information can be used to apply the methodology, and implement the parallelization tool, in the context of other programming languages. Currently we are working on a refactoring tool-chain for Scala. This language has a concurrency tool-set (Akka) very similar to that of Erlang. Therefore, the main ideas presented in this paper can be reused in the parallelisation of Scala programs – though many interesting problems remain due to the differences in the two languages. Even more problems can arise when targeting other programming languages (e.g. OpenMP would be a logical choice).

This presentation focused on a set of well-known parallel patterns, including the task farm, the pipeline and the divide-and-conquer patterns. We explained what kind of program fragments (pattern candidates) can be automatically identified by PaRTE, and how to select the most promising candidates. The selected candidates can be refactored into applications of an algorithmic skeleton library, which implements the parallel patterns. Many of the refactoring transformations can be performed automatically by the tool, though the most complex ones can only be carried out by hand currently. We have evaluated the applicability of the methodology on a number of real-world, open source code bases, and found that the tool is indeed able to identify many interesting parallel pattern candidates.

There are still, of course, some shortcomings and unsolved implementation issues. We do not claim that our parallel pattern candidate discovery is either complete or even sound. Some of the discovery rules are trade-offs between soundness

and (efficient) computability. It might happen that the tool presents a "fake candidate", which cannot be refactored into a pattern. This should not be a problem, however, since our tool is not a compiler optimization running unsupervised. Fake candidates can be discarded by the user after proper consideration.

The completeness of the discovery is not guaranteed either. Our rules can find many interesting candidates, but there may be many more: in the future the rule set can be extended, or can be made more accurate. What is even more important is the good ranking of the candidates. Completely automatic performance estimations are not feasible in more complex cases, so either good heuristics are required (such as our existing one using some complexity metrics), or the integration of our tool with an environment supporting development-time experimentation and dynamic analyses.

Automatic transformations also have a number of unsolved issues. It might happen that a candidate is found by the discovery, but the current definition of the appropriate transformation is not able to cover that specific candidate. In this case either manual transformation of the code is needed (which is error-prone) or the application of a sequence of built-in transformations including program shaping before, and code clean-up after, the introduction of the desired pattern (like in the case of the *d&c* pattern). We are working actively to ensure, and also to prove, soundness of the refactoring transformations – clearly, this is a crucial requirement for our approach.

Acknowledgement

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013) in 2017-2018, by the Hungarian Government through the New National Excellence Program of the Ministry of Human Capacities in 2016-2017, by the Higher Education Restructuring Fund of the Hungarian Government in 2015-2016, and by the Seventh Framework Programme (FP7) of the European Union under the contract number 288570 in 2013-2015.

REFERENCES

[1] Marco Aldinucci, Massimo Coppola, and Marco Danelutto. 1998. Rewriting Skeleton Programs: How to Evaluate the Data-Parallel Stream-Parallel Tradeoff. In *Proc of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, S. Gorlatch (Ed.). Fakultät für mathematik und informatik, Uni. Passau, Germany, Germany, 44–58. http://www.di.unipi.it/~aldinuc/paper_files/1998_transf_cmpp.pdf

[2] Joe Armstrong. 2013. *Programming Erlang*. The Pragmatic Bookshelf, 548 pages, ISBN 978.1-93778-533-6, NC, USA.

[3] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. 2012. A Scalability Benchmark Suite for Erlang/OTP. In *Proc. of 11th ACM SIGPLAN workshop on Erlang*. ACM, New York, NY, USA, 33–42. http://doi.acm.org/10.1145/2364489.2364495

[4] Stavros Aronis and Konstantinos Sagonas. 2013. On Using Erlang for Parallelization. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 295–310.

[5] Olivier Boudeville, Francesco Cesarini, Natalia Chechina, Kenneth Lundin, Nikolaos Papaspyrou, Konstantinos Sagonas, Simon Thompson, Phil Trinder, and Ulf Wiger. 2013. *Trends in Functional Programming: 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter RELEASE: A High-Level Paradigm for Reliable Large-Scale Server Software, 263–278. https://doi.org/10.1007/978-3-642-40447-4_17

[6] István Bozó, Viktoria Fördős, Dániel Horpácsi, Zoltán Horváth, Tamás Kozsik, Judit Kőszegi, and Melinda Tóth. 2015. Refactorings to Enable Parallelization. In *Trends in Functional Programming*, Jurriaan Hage and Jay McCarthy (Eds.). Springer International Publishing, Berlin, Heidelberg, 104–121. https://doi.org/10.1007/978-3-319-14675-1_7

[7] István Bozó, Viktoria Fördős, Zoltán Horváth, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Kőszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. 2014. Discovering Parallel Pattern Candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang (Erlang '14)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/2633448.2633453

[8] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, and M Tóth. 2011. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*. Proceedings of the Estonian Academy of Sciences, Tallin, Estonia, 138–148.

[9] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. 2014. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming* 42, 4 (01 Aug 2014), 564–582. https://doi.org/10.1007/s10766-013-0266-5

[10] C. Brown, V. Janjic, K. Hammond, H. Schöner, K. Idrees, and C. Glass. 2014. Agricultural Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, NJ, USA, 36–43. https://doi.org/10.1109/PDP.2014.94

[11] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. 2012. ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. In *Trends in Functional Programming*, Ricardo Peña and Rex Page (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 82–97.

[12] R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (1977), 44–67. https://doi.org/10.1145/321992.321996

[13] Murray Cole. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA.

[14] Murray Cole. 2004. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Comput.* 30, 3 (March 2004), 389–406. http://dx.doi.org/10.1016/j.parco.2003.12.002

[15] J. Darlington, Y. Guo, Y. Jing, and H. W. To. 1995. Parallel Skeletons for Structured Composition. In *Proc. of the 15th Symposium on Principles and Practice of Parallel Programming*. ACM New York, NY, USA, 19–28.

[16] Danny Dig. 2011. A Refactoring Approach to Parallelism. *IEEE Softw.* 28 (2011), 17–22. Issue 1. https://doi.org/10.1109/MS.2011.1

[17] Danny Dig, John Marrero, and Michael D. Ernst. 2009. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 397–407. https://doi.org/10.1109/ICSE.2009.5070539

[18] Danny Dig, John Marrero, and Michael D. Ernst. 2011. How Do Programs Become More Concurrent: A Story of Program Transformations. In *Proceedings of the 4th International Workshop on Multicore Software Engineering (IWMSE '11)*. ACM, New York, NY, USA, 43–50. https://doi.org/10.1145/1984693.1984700

[19] Faculty of Informatics ELTE. 2017. RefactorErl Home Page. (2017). http://plc.inf.elte.hu/erlang/.

[20] Faculty of Informatics ELTE. 2017. Wiki page of the ParaPhrase Refactoring Tool for Erlang. (2017). http://pnyf.inf.elte.hu/trac/refactorerl/wiki/parte.

[21] B. Freisleben and T. Kielmann. 1995. Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *Computing and Informatics* 14 (1995), 579–596. Issue 6.

[22] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. 2009. Profiling Java Programs for Parallelism. In *Proc.*

IWMSE '09. IEEE Computer Society Washington, DC, USA, 49–55. https://doi.org/10.1109/IWMSE.2009.5071383

[23] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. 2013. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *Formal Methods for Components and Objects*. Lecture Notes in Computer Sscience, Vol. 7542. Springer, Berlin, Heidelberg, 218–236.

[24] Fred Hebert. 2013. *Learn You Some Erlang for Great Good!* No Starch Press, San Francisco, CA 94103 USA.

[25] Dániel Horpácsi, Judit Kőszegi, and Simon Thompson. 2016. Towards Trustworthy Refactoring in Erlang, In Fourth International Workshop on Verification and Program Transformation, Geoff Hamilton, Alexei Lisitsa, and Andrei P. Nemytykh (Eds.). *Electronic Proceedings in Theoretical Computer Science* 216, 83–103. http://kar.kent.ac.uk/56750/

[26] Tamás Kozsik, Melinda Tóth, István Bozó, and Zoltán Horváth. 2017. Static analysis for divide-and-conquer pattern discovery. *Computing and Informatics* 35 (2017), 764–791. Issue 4.

[27] Tamás Kozsik, Melinda Tóth, and István Bozó. 2018. Free the Conqueror! Refactoring divide-and-conquer functions. *Future Generation Computer Systems* 79, Part 2 (2018), 687 – 699. https://doi.org/10.1016/j.future.2017.05.011

[28] Lapedo. 2014. Available at http://lapedo.weebly.com/. (2014).

[29] Huiqing Li and Simon Thompson. 2008. Tool support for refactoring functional programs. In *WRT '08: Proceedings of the 2nd Workshop on Refactoring Tools*. ACM, New York, NY, USA, 1–4. https://doi.org/10.1145/1636642.1636644

[30] M. Logan, E. Merritt, and R. Carlsson. 2010. Erlang and OTP in Action. Manning Publications Co., ISBN 9781933988788., New York, USA.

[31] Rita Loogen. 2012. Eden – Parallel Functional Programming with Haskell. In *Central European Functional Programming School*, Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer (Eds.). Lecture Notes in Computer Science, Vol. 7241. Springer, Berlin Heidelberg, 142–206. https://doi.org/10.1007/978-3-642-32096-5_4

[32] Jonathan Mak, Karl-Filip Faxén, Sverker Janson, and Alan Mycroft. 2010. Estimating and Exploiting Potential Parallelism by Source-level Dependence Profiling. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I (EuroPar'10)*. Springer-Verlag, Berlin, Heidelberg, 26–37. http://dl.acm.org/citation.cfm?id=1887695.1887700

[33] Shane A. Markstrum and Robert M. Fuhrer. 2009. Extracting Concurrency via Refactoring in X10, in Proceedings of the 3rd ACM Workshop on Refactoring Tools. (2009). http://refactoring.info/WRT09/#program

[34] Shane A. Markstrum, Robert M. Fuhrer, and Todd D. Millstein. 2009. Towards Concurrency Refactoring for x10. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 303–304. https://doi.org/10.1145/1504176.1504226

[35] Simon Marlow. 2012. Parallel and Concurrent Programming in Haskell. *Central European Functional Programming School* 7241 (2012), 339–401. http://community.haskell.org/~simonmar/papers/par-tutorial-cefp-2012.pdf

[36] Greg Michaelson, Andrew Ireland, and Peter King. 1997. Towards a Skeleton Based Parallelising Compiler for SML. In *Proceedings of 9th International Workshop on Implementation of Functional Languages*. IFL'97, St. Andrews, Scotland, UK, 539–546.

[37] K. Molitorisz. 2013. Pattern-Based Refactoring Process of Sequential Source Code. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, Genova, Italy, 357–360. https://doi.org/10.1109/CSMR.2013.49

[38] Korbinian Molitorisz, Jochen Schimmel, and Frank Otto. 2012. Automatic Parallelization Using Autofutures. In *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT'12)*. Springer-Verlag, Berlin, Heidelberg, 78–81. https://doi.org/10.1007/978-3-642-31202-1_8

[39] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., Burlington, Massachusetts.

[40] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999, corrected 2005. *Principles of Program Analysis*. Springer, Berlin, Heidelberg.

[41] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation. Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA.

[42] M. Pitidis and K. Sagonas. 2011. Purity in Erlang. Proc. 22nd Int'l Conf. on Implementation and Application of Functional Languages, *Lecture Notes in Computer Science,Springer Berlin, Heidelberg* 6647 (2011), 137–152.

[43] Skel Tutorial. 2014. Available at http://chrisb.host.cs.st-andrews.ac.uk/skel-test-master/tutorial/bin/tutorial.html. (2014).

[44] The ParaPhrase project. 2014. http://www.paraphrase-ict.eu. (2014).

[45] Melinda Tóth and István Bozó. 2012. Static Analysis of Complex Software Systems Implemented in Erlang. *Central European Functional Programming School* 7241 (2012), 440–498.

[46] AHG University. 2014. Multi Agent System. (2014). https://github.com/ParaPhraseAGH/erlang-mas/tree/0.1.

[47] Uppsala University. 2017. The DIALYZER: a DIscrepancy AnaLYZer for ERlang programs. (2017). http://www.it.uu.se/research/group/hipe/dialyzer.

[48] Jan Wloka, Manu Sridharan, and Frank Tip. 2009. Refactoring for Reentrancy. In *ESEC/FSE '09*. ACM, Amsterdam, 173–182. https://doi.org/10.1145/1595696.1595723

[49] WP2. 2013. *Final Pattern Definition Report*. Technical Report. University of Pisa. http://paraphrase-ict.eu/Deliverables/deliverable-2.5/deliverable-2.5-report/view

[50] WP2. 2013. *Initial Implementation of Application-Specific Patterns*. Technical Report. University of Pisa. http://paraphrase-ict.eu/Deliverables/deliverable-2.6/deliverable-2.6-report/view

[51] V. Zsók, Z. Hernyák, and Z. Horváth. 2006. Designing Distributed Computational Skeletons in D-Clean and D-Box. Central European Functional Programming School, Lecture Notes in Computer Science, Springer 4146 (2006), 223–256.

Refactorings to Enable Parallelization*

Bozó, I.[2], Fördős, V.[2], Horpácsi, D.[1], Horváth, Z.[2],
Kozsik, T.[1], Kőszegi, J.[1], Tóth, M.[2]

[1] Eötvös Loránd University, Budapest, Hungary
[2] ELTE-Soft Nonprofit Ltd., Budapest, Hungary
{bozoistvan,f-viktoria,daniel-h,hz,kto,koszegijudit,tothmelinda}@elte.hu

Abstract. We propose program analyses to identify parallelizable code fragments, and program transformations to change those fragments into applications of high-level parallel patterns. The methodology has been worked out, and is presented here, in the context of the Erlang programming language, but the approach is applicable in other languages as well.

1 Introduction

Refactoring is the process of restructuring, shaping or transforming a program in order to improve its quality, to change its non-functional properties or to make it suitable to add a new feature. This activity can be carried out by hand, or by using program transformation tools. One possible application area of refactoring is the introduction of parallelism into existing sequential programs, which is the main focus of this paper. When parallelizing industrial-scale software applications, a tool can provide invaluable help in decision making as well as in the semi-automatic application of refactoring transformations. Such a tool should offer guidance to its user on what refactoring decisions are to be made, on where it is the most fruitful to introduce parallelism, and on how to achieve the desired program structure.

The EU ParaPhrase project [1] proposes a novel structured design and implementation process for parallel programming, where developers exploit a variety of high-level parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically re-mapped to meet application needs and hardware availability. Among others, the project aims to produce new tools and techniques that can: i) (semi-)automatically locate suitable pattern candidates in Erlang programs; and ii) recommend transformed versions of these pattern candidates that yield significant speedup on a given parallel architecture. The refactoring tool PaRTE (viz. ParaPhrase Refactoring Tool for Erlang) [2] identifies pattern candidates in Erlang programs. It applies static analyses to determine which operations are performed element-wise on multiple (perhaps many) data items, and whether

* This work has been supported by the European Union Framework 7 under contract no. 288570. ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multi-core System

these operations may be executed in parallel without changing the semantics of the program. With PaRTE, the programmer performs a series of transformations that turn the original program into an equivalent, but effectively parallelized code – Skel [3], an algorithmic skeletons library for Erlang, is used to describe parallelism. PaRTE offers many refactoring transformations, which can be executed under the supervision, and control, of the software developer. The prototype implementation of PaRTE will be made available for the wider public on the web page of the ParaPhrase project [4].

In this paper we show how to identify various syntactic forms that can effectively be transformed into parallel code. We do not go into details on the side conditions of transformations, which are meant to ensure that the parallelization preserves the order of side effects; the interested reader is guided to [2]. Due to space restrictions, we focus on the *task farm* skeleton. Our main novel contributions are the following:

1. a method to automatically identify structures in a program text that can be replaced with applications of algorithmic skeletons;
2. transformations to enable parallelization for a wide range of syntactic forms;
3. formal rules to substantiate the implementation.

The techniques presented in the following sections have been defined and applied on Erlang/OTP applications, but can be adapted to other languages as well.

The rest of the paper is structured as follows. In Section 1.1, the notion of algorithmic skeletons is briefly presented. Section 2 provides an overview of our methodology and a simple example. Section 3 explains how the general concepts of control-flow and data-flow analyses are adapted for Erlang. These analyses are used in Section 4, which describes the pattern identification algorithms, as well as the related transformation rules. Section 5 discusses related work, and Section 6 concludes the paper.

1.1 Algorithmic Skeletons

Algorithmic skeletons [5] are well-designed, frequently used patterns of parallelism that aim to help programmers focus on the application logic of the software, and to avoid dealing with low level details of parallel execution, such as synchronization and liveness properties. Several libraries provide algorithmic skeleton implementations for different languages. For Erlang, the Skel [3] library can be used. Skel supports the `pipe`, `farm`, `ord`, `reduce`, `map` and `feedback` skeletons; in addition, it uses `seq` as a primitive pattern to wrap sequential computations.

To illustrate the use of this library, consider the following example. Assume we have a list of modules, and we want to produce the syntax-tree for each of them. We should perform the same sequence of operations (read, scan and parse) on each module:

```
1 [ parse(scan(read(Module))) || Module <- Modules ]
```

We could perform this chain of operations on each module independently, as separate processes, and execute the composition of the 3 functions as a pipeline. Therefore, we can introduce a farm skeleton (with e.g. 10 workers) to process the elements of the module list, and turn the function composition into stages of a pipeline.

```
1  skel:do( [ { farm, [ { pipe, [ { seq, fun read/1 },
2                                 { seq, fun scan/1 },
3                                 { seq, fun parse/1 }
4                               ] }
5              ], 10 } ], Modules ).
```

2 Overview: where and how to parallelize

Refactoring transformations change the structure of source code without changing its meaning. That is, such transformations do not alter the semantics of the program, but they typically affect some non-functional properties, such as size, complexity, speed or memory footprint. Parallelization, as a refactoring step, usually increases program size and code complexity, but reduces the execution time, while it preserves the observable behaviour.

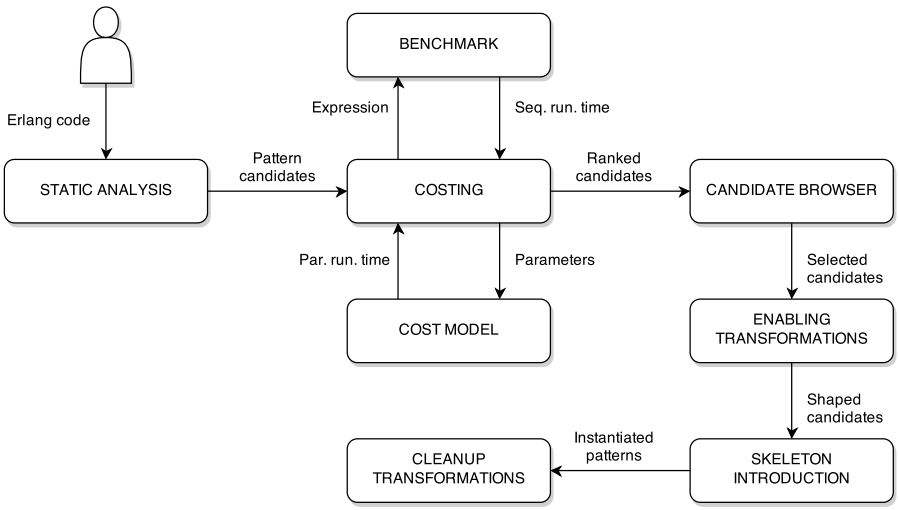


Fig. 1. Overview of the parallelization method

Refactorings are usually implemented as (series of) graph rewritings of the abstract syntax tree (the model) of the program. In most cases, these graph transformations are context-sensitive, targeting particular parts of the syntax tree, changing them according to some well-defined rewriting rules. Usually,

code fragments to be restructured are selected by the user. In contrast, in our approach, the possible targets of parallelization are automatically identified; namely, we collect every occurrence of the syntactic pattern, perform semantics-based filtering, and then order the candidates.

2.1 Where to parallelize

Pattern discovery locates all the subtrees (program fragments) that are amenable for parallelization, that is, it spots the syntactic subtrees that i) are compatible with the syntactic scheme of a rule, and ii) fulfil additional requirements that cannot be expressed with the syntactic scheme (non-structural, semantic requirements, such as predicates on attributes of a node).

In Figure 1 you can overview our parallelization process. You can see that the very first step is the static analysis of the code, including the discovery of pattern candidates. This phase takes the source code of the program, and yields a set of sequential code fragments that implement element-wise computations that could be implemented with well-known patterns such as farms, maps or pipelines. These are all candidates for parallelization, that is, in principle, they could be turned into instances of parallel skeletons; in practice, only a few of them are worth being transformed. It is possible that a candidate implements not a single pattern, but a combination of multiple patterns, which still falls into the kind of structured parallelism, although it might need some really complex refactoring steps to achieve.

Example. Let us consider a small piece of Erlang code (Listing 2.1) implementing sparse matrix-vector multiplication.

```
1 vxv(Row, Col) -> ... % sparse vector-vector multiplication
2
3 mxv(Rows, Col) ->
4   Product = [{I, vxv(Row, Col)} || {I, Row} <- Rows],
5   lists:filter(fun({_, V}) -> V /= 0 end, Product).
```

In this example, pattern discovery would spot a *farm* candidate: the list comprehension realising the multiplication of each matrix row with the vector (line 4) could be turned into a parallel computation using a task farm. Indeed, it invokes the same computation on each and every row, which implements an independent (and therefore parallelizable) set of computations.

Costing and ranking In legacy Erlang code, the discovery phase might identify dozens of parallelizable computations. However, not all the candidates are worth being parallelized, since the communication overhead may be nearly as high as the performance improvement gained by parallel execution. Therefore, we apply dynamic analysis and measure the execution time of the kernel computation, then apply machine-calibrated, pattern-specific cost models [6,7], in order to forecast the possible performance improvement, and rank the candidates accordingly.

2.2 How to parallelize

As mentioned already, refactoring transformations are realised as rewritings of the abstract model of the program. In our approach, the process of parallelization is decomposed into smaller but simpler steps, which can be categorized into three main groups: enabling steps, skeleton introduction and cleanup steps.

Enabling transformations Conceptually, rewriting to parallel skeletons is only applicable in the case of a limited number syntactic forms. The reason for this is twofold: i) we want the skeleton introduction to be extremely simple (i.e. error-free) and ii) we want the source code to enable easy check of side conditions. Therefore, we define preliminary, enabling transformations that trustworthily reshape parallel pattern candidates into such simple forms of code (see section 4.1).

To see an example, let us go back to the previous example shown in Listing 2.1. In order to be able to turn the list comprehension into an instance of a parallel skeleton, we need to simplify the head of the comprehension. We use the most general list comprehension rewriting, which extracts the generator as well as the head of the comprehension.

```
1 mxv(Rows, Col) ->
2   Product =
3     begin
4       L = [{I, Row} || {I, Row} <- Rows],
5       F = fun({I, Row}) -> {I, vxv(Row, Col)} end,
6       [F(X) || X <- L]
7     end,
8   filter(fun({_, V}) -> V /= 0 end, Product).
```

Skeleton introduction This phase is about introducing parallel skeletons in place of (possibly already simplified) list comprehensions. Technically, we rewrite the sequential implementation into an instance of the parallel skeleton. Side effects should be checked here in order to guarantee behaviour-preservation (see some details in Section 4.1).

```
1 mxv(Rows, Col) ->
2   Product =
3     begin
4       L = [{I, Row} || {I, Row} <- Rows],
5       F = fun({I, Row}) -> {I, vxv(Row, Col)} end,
6       skel:do([{farm,{seq, fun(X)-> F(X) end},10}], L)
7     end,
8   filter(fun({_, V}) -> V /= 0 end, Product).
```

Cleanup transformations Enabling transformations and the instantiation of skeletons yield a parallel version of the very same computation; however, they may introduce undesired and unnecessary complexity into the source code. A

refactoring process, after all, should improve the readability and the maintainability of the software. It is valuable that the program became faster, but we also need to clean up the syntactic noise we introduced during the first two phases.

There can be various refactoring steps that tidy up Erlang code. We focussed on those ones that are potentially applicable after executing some enabling transformations, so that they can remove the complexity introduced but not really needed.

Also, it is worth mentioning that the refactoring steps defined in this paper, including both the enabling and the cleanup transformations, are generally applicable for any Erlang project, not only for parallelization purposes. However, it can also happen that even if the pattern discovery suggest the parallelization of a code fragment, none of the above defined enabling steps are applicable.

Back to the example. The skeleton introduction rule, by definition, enclosed the `F(X)` expression into a fun-expression. However, it is apparent that there is no need for this abstraction. One possible cleanup transformation is eliminating such unnecessary fun-expressions.

```
1 mxv(Rows, Col) ->
2   Product =
3     begin
4       L = [{I, Row} || {I, Row} <- Rows],
5       F = fun({I, Row}) -> {I, vxv(Row, Col)} end,
6       skel:do([{farm,{seq, F},10}], L)
7     end,
8   filter(fun({_, V}) -> V /= 0 end, Product).
```

Another needless complication was introduced by the enabling transformation. Namely, the begin-end block around the matrix-vector product computation can be eliminated quite simply. In general, this cleanup transformation is an iterative process: we enlarge the block until it becomes a top-level expression (or it contains only one single expression), and then remove it.

```
1 mxv(Rows, Col) ->
2   L = [{I, Row} || {I, Row} <- Rows],
3   F = fun({I, Row}) -> {I, vxv(Row, Col)} end,
4   Product = skel:do([{farm,{seq, F},10}], L),
5   filter(fun({_, V}) -> V /= 0 end, Product).
```

Finally, in some cases, we can also remove the list comprehension composing the list of the elements to be processed: if in the original list comprehension only contained a generator (and the original list is homogeneous), we can freely eliminate the variable and the extra list comprehension, putting the original list into the `skel:do/2` call.

```
1 mxv(Rows, Col) ->
2   F = fun({I, Row}) -> {I, vxv(Row, Col)} end,
3   Product = skel:do([{farm,{seq, F},10}], Rows),
4   filter(fun({_, V}) -> V /= 0 end, Product).
```

3 Supporting Analyses

Although the initial selection of pattern candidates is syntax based, a thorough static semantic analysis is required to filter out candidates that can be transformed automatically. In this section we briefly introduce the used analyses and notations.

3.1 Control Flow Graph

The interprocedural Control Flow Graph of an Erlang program is labelled directed graph represented by the $CFG = (V, E)$ pair where V is the set of nodes containing the subexpressions of the analysed program and some special nodes; and E is the set of graph edges representing the relation an expression evaluated after an other expression.

For all $v \in V$ exactly one of the following statements holds:

- v is a subexpression of the program
- v is a special node representing the starting point of a function ($start_f$)
- v represents the end of a function evaluation (end_f)
- v represents the call of a function ($call_f$)
- v represents the return of a function call (ret_f)

For all $(u, l, v) \in E$ edge exactly one of the following statements holds:

- u and v represent subexpressions and v is evaluated directly after u when the condition l holds
- $u = call_f$ represents the call of a function and $v = start_f$ is the starting point of the same function
- $u = end_f$ represents the end of a function evaluation and $v = ret_f$ is the return point of the call of the same function

Here we use the notation f to identify a function. In Erlang it is represented by a module name, function name and arity triple. If there is no condition between the evaluation of the two expressions then the label is empty.

$EP(v)$ denotes the set of execution paths in the CFG starting from the node $v \in V$.

3.2 Data Flow Graph

The interprocedural Data Flow Graph of an Erlang program is a labelled directed graph represented by the $DFG = (N, L)$ pair where N is the set of nodes containing the subexpressions of the analysed program; and L is the set of graph edges representing the relation that the value of an expression can flow to an other expression, or can be a part of it.

For all $(u, l, v) \in L$ edge exactly one of the following statements holds:

- $l = flow$ and the value of the expression u flows to the expression v

- $l = c_i$, v is a tuple and u is the i^{th} element of v
- $l = s_i$, u is a tuple pattern and v is the i^{th} element of the pattern
- $l = c_h$, v is a list and u is element of the list
- $l = s_h$, u is a list pattern and v is element of the pattern
- $l = c_t$, v is a list and u is the tail of the list
- $l = s_t$, u is a list pattern and v is the tail of the pattern
- $l = d$, the value of v depends on the value of u

To calculate whether the value of a certain expression can reach an other expression we define the *data flow reaching* relation. The zeroth order data-flow reaching relation ($\overset{\mathbf{0f}}{\rightsquigarrow}$) is the minimal relation that satisfies the following rules:

$$n \overset{\mathbf{0f}}{\rightsquigarrow} n \qquad \text{(reflexive)} \qquad\qquad \frac{n_1 \overset{\mathbf{c_i}}{\rightarrow} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3,\ n_3 \overset{\mathbf{s_i}}{\rightarrow} n_4}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_4} \qquad \text{(c-s rule)}$$

$$\frac{n_1 \overset{\mathbf{flow}}{\rightarrow} n_2}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2} \qquad \text{(flow rule)} \qquad\qquad \frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ n_2 \overset{\mathbf{0f}}{\rightsquigarrow} n_3}{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_3} \qquad \text{(transitive)}$$

A detailed and more precise definition of the DFG and the reaching relation can be found in [8], where the context dependent *first-order data flow reaching* is also defined. The so-called *compact data flow reaching* is defined as follows:

$$\frac{n_1 \overset{\mathbf{0f}}{\rightsquigarrow} n_2,\ \nexists n_0,\ n_0 \neq n_1 \colon\ n_0 \overset{\mathbf{0f}}{\rightsquigarrow} n_1}{n_1 \overset{\mathbf{0f_c}}{\rightsquigarrow} n_2} \qquad \text{(compact)}$$

We also define the dependence relation ($\overset{\mathbf{dep}}{\rightsquigarrow}$) over the *Dependence Graphs* (DG) of Erlang programs [8]. We calculate the DG based on the control and data flow graphs by eliminating the unnecessary sequencing from the CDG and adding the data relation based on the DFG.

4 Identifying Candidates and Enabling Transformations

This section defines various refactoring steps that can contribute to the parallelization process. These are called enabling transformations, which reshape programs into forms compatible with the skeleton introduction rules. Reshaping is formally defined by term rewriting rules. Let us introduce the following notation:

$$\frac{A}{\blacktriangledown\ B} \quad \text{WHEN } C$$

Here, A is called the (syntactic) pattern, and B is called the replacement. They are both syntactic schemes. Namely, A and B may contain *metavariables*, and also there are special metavariables (e.g. $\overline{E}$) that can match and record multiple syntactic elements in a row (such as expression sequences, parameter lists), embodied as siblings in the syntax tree. In addition, the formalism supports

the concept of ***fresh variables*** (unused names in the scope), which can be used to introduce new function and variable names. Finally, a rule can have **parameters**, which have to be provided each time the rule is applied.

The notation above means that A can be rewritten to B if the condition C holds. A as well as B can be either expressions or functions, and the replacement term is allowed to be composed of multiple terms. If the replacement is given as an expression sequence, it will be enclosed by a begin-end block in order to guarantee syntax-validity.

The WHEN clause, which is defining the context-dependent side-conditions, is a list of first-order logic formulae, each needed to hold for the rule to be applicable. The formulae typically rely on semantic predicates, context-dependent relations of the program entities mentioned in the syntactic schemes, including scopes, references, control-flow and data-flow. Note that the formulae may also refer to metavariables bound in the replacement term: namely, the synthesised code fragment can be influenced by the condition. For this reason, the WHEN condition may also contribute to the post-condition of the refactoring rule.

Compound rules Complex, extensive refactoring steps are composed of multiple rewriting rules. The individual rewritings in this case need not (and usually do not) preserve the behaviour, only their proper composition yields a refactoring. For the moment, we introduce one combinator (the sequential composition), and one rule modifier (scope restriction) for composing and refining rules. In the formalism, the rule composition is denoted with the THEN keyword, while the scope of the rule is restrited by the metavariable following the IN keyword.

$$\frac{A}{B_1 B_2} \quad \text{WHEN } C$$

$$\text{THEN IN } B_2 : \quad \frac{D}{E} \quad \text{WHEN } F$$

The above example expresses that the "nested" rule (D to E if F) is applied after executing the outer rule, and replacements with the inner rule are only made in the subexpressions of B_2. If B_2 has been removed by the outer rule, the inner rule does not have any effects.

4.1 List Comprehensions

A list comprehension expression is a good candidate for introducing a farm skeleton, as it implements elementwise processing. With some further analyses, it is also decidable whether a farm candidate is suitable for introducing a pipeline: the head expression of the list comprehension can be reshaped to a function composition.

Introducing a skeleton It is allowed to introduce a farm skeleton directly when the list comprehension fulfils the following requirements: i) the head expression is an invocation of a unary function, ii) it has exactly one generator and does not have any filters, and iii) the argument of the function call and the pattern in the generator are references to the same variable (we do not allow compound patterns as they may implement filtering). If the above conditions are met, we can straightforwardly compose the worker function of the farm and rewrite [6] the comprehension to the skeleton[3]. (Actually, we also need the worker to be side effect free, but we guide the reader to [2] for further details.)

The following rule formally defines farm skeleton introduction. In the rewrite rule, F may match either an identifier or a function expression, therefore the application gets wrapped in order to handle both cases properly. Note that **Nw** (the number of parallel workers) is a parameter of the transformation.

$$\frac{[\ F(P)\ ||\ P \leftarrow \mathit{List}\ ]}{\mathrm{skel}:\mathrm{do}([\{\mathrm{farm}, [\{\mathrm{seq}, \mathbf{fun}(P) \rightarrow F(P)\ \mathbf{end}\}], \mathbf{Nw}\}], \mathit{List})} \quad \text{WHEN} \quad \begin{array}{l}\mathrm{var}(P),\\ \mathrm{pure}(F)\end{array}$$

Preparing the list List comprehensions not meeting the former criteria need to be prepared for the introduction of parallel skeletons. The shaping is realised by *enabling transformations*. For example, when we have a unary function call in the head of the list comprehension, but the generator have a compound pattern that can do some filtering on the list, or when we have multiple generators or filters, we have to pre-generate the input data list. We pick the argument of the function call and use it as a head expression in a new list comprehension. The replacement list comprehension processes this pre-generated list, matches the argument to each element, and calls the original function on these. We get a functionally equivalent code, but now it supports the parallel refactoring.

Note that F can match arbitrarily compound expressions, so that we need to exclude those cases when F refers to variables bound by the generators (otherwise we would produce invalid code suffering from references to unbound variables).

$$\frac{[\ F(P)\ ||\ \overline{GFs}\ ]}{\begin{array}{l}\boldsymbol{List} = [\ P\ ||\ \overline{GFs}\ ],\\ [\ F(P)\ ||\ P \leftarrow \boldsymbol{List}\ ]\end{array}} \quad \text{WHEN} \quad \begin{array}{l}|\ \overline{GFs}\ | \geq 1,\\ |\ \mathrm{boundVars}(\overline{GFs}) \cap \mathrm{freeVars}(F)\ | = 0\end{array}$$

Example. We have a function definition containing a list comprehension with multiple generators.

```
1 f(List1,List2) ->
2   [g({X,Y}) || X <- List1, Y <- List2].
```

We cannot apply the farm introduction transformation, since it does not match to the corresponding rule. Therefore, we need to prepare our code for parallelization: apply the previously introduced enabling transformation rule.

[3] The *farm* skeleton itself is not guaranteed to preserve the order of list elements, one might need to apply an additional *ord* skeleton to preserve ordering.

```
f(List1,List2) ->
  begin
    List = [{X,Y} || X <- List1, Y <- List2],
    [g({X,Y}) || {X,Y} <- List]
  end.
```

General rule We can also distinguish cases where we need to fix the head expression of the list comprehension. However, we can cover all of these cases with one general rule. This matches arbitrary head expressions and an arbitrary number of generators/filters in a list comprehension. Then the enabling transformation collects all of the variables that are bound by the generators and needed for evaluating the head expression. We compose a tuple from these variables and by using the tuple as a head expression of a new list comprehension, we pre-generate a list with the original generators and filters. We wrap the original head expression with a unary anonymous function having the formerly mentioned tuple in its argument. The resulting list comprehension processes the pre-generated list and invokes the wrapper function for each element.

$$\frac{[\ \mathit{Head}\ ||\ \overline{\mathit{GFs}}\ ]}{\begin{array}{l}\boldsymbol{\mathit{List}} = [\ \{\overline{\mathit{Vars}}\}\ ||\ \overline{\mathit{GFs}}\ ],\\ \boldsymbol{\mathit{Fun}} = \textbf{fun}\ (\{\overline{\mathit{Vars}}\})\ \rightarrow\ \mathit{Head}\ \textbf{end},\\ [\ \boldsymbol{\mathit{Fun}}(\boldsymbol{\mathit{X}})\ ||\ \boldsymbol{\mathit{X}} \leftarrow \boldsymbol{\mathit{List}}\ ]\end{array}} \quad \text{WHEN} \quad \begin{array}{r}\overline{\mathit{Vars}} = \text{boundVars}(\overline{\mathit{GFs}})\\ \cap\ \text{freeVars}(\mathit{Head})\end{array}$$

4.2 Library Calls

Erlang/OTP libraries contain plenty of functions that realise elementwise processing on different kinds of data structures. Thus, the application of these functions are candidates for parallelization. Since our parallel skeletons only work with lists, enabling transformations are responsible for transforming such data structures to lists; in addition, these library calls have to transformed to semantically equivalent expressions or expression sequences that contain list comprehensions fulfilling the formerly defined requirements for parallelization. As result, we will be able to introduce a skeleton in place of the list comprehension.

In this section, we focus on identifying library function calls for which we can define enabling transformation rules that prepare the introduction of the farm skeleton, and we highlight some example rules.

Functions of the *lists* module The most obvious farm candidate is any application of the `lists:map` function, as it can be trivially transformed to an appropriate list comprehension.

$$\frac{\textbf{lists:map}(\mathit{F}, \mathit{List})}{[\ \mathit{F}(\boldsymbol{\mathit{X}})\ ||\ \boldsymbol{\mathit{X}} \leftarrow \boldsymbol{\mathit{List}}\ ]}$$

Almost the same transformation can be applied in the case of `flatmap`, the only additional step is to flatten the resulted list. Furthermore, we have identi-

fied other library functions calls to which we define non-trivial enabling transformations that prepare them for farm introduction. For example, in the case of `filter`, we construct a function that turns *filtering* into *mapping*: the worker returns a single-element list for elements accepted by the original filter, and an empty list otherwise; finally, we append the results. With a similar method we can handle the calls to the `filtermap` function, too.

In the case of function calls to `zipWith` (`zipWith3` goes similarly), first we have to use the ordinary `zip` to construct pairs of elements, and reshape (wrap) the original binary function so that it takes tuples rather than two separate elements.

lists:zipWith(*F*, *List1*, *List2*)

List = **lists:zip**(*List1*, *List2*),
Fun = **fun** ({***Arg1***, ***Arg2***}) → *F*(***Arg1***, ***Arg2***) **end**,
[***Fun***(***X***) || ***X*** ← ***List***]

Processing other iterable data structures In Erlang we have several data structures of which library offers `to_list` and `from_list` functions for conversions. We identify calls for `map`, `filter` and some other elementwise processing functions of `dict`, `orddict`, `sets` and `ordsets` libraries as farm candidate and offer enabling transformations for them. The following rule shows the transformation belonging to `dict:map` function call. Here we convert the dictionary to a list containing key-value pairs, do the actual list processing inside a list comprehension and then convert back the resulted list to dictionary.

dict:map(*F*, *Dict*)

List = **dict:to_list**(*Dict*),
Fun = **fun** ({***Arg1***, ***Arg2***}) → {***Arg1***, *F*(***Arg1***, ***Arg2***)} **end**,
Result = [***Fun***(***X***) || ***X*** ← ***List***],
dict:from_list(***Result***)

4.3 Map-like functions

A programmer can write a function that processes a list element-wise in various ways. List comprehensions and standard library functions processing streams of video frames like `lists:map/2` provide syntactically pleasing notations to express such computations. A programmer, especially an inexperienced one, might also decide to write a recursive function to iterate over the elements of a list. While in general this is not the preferred style, it may actually yield readable code if the computation to apply on each list element is complex enough to look ugly inside the head of a list comprehension, and the programmer wants to avoid the performance overhead incurred by the use of higher-order functions such as `lists:map/2`. Note that Erlang does not support partial application, and this,

as well as the syntactic noise needed for anonymous functions, make the use of `lists:map/2` less appealing.

The ability to automatically identify map-like recursive functions is advantageous for a number of reasons. Firstly, when tidying up the code written by an inexperienced programmer, one can find and remove unnecessary recursive functions by replacing their use with equivalent list comprehensions. Secondly, one can turn a map-like recursive function into the composition of two tail-recursive ones, and hence enable tail-call elimination. (This may be important in languages where tail recursion is, but *tail recursion modulo cons* is *not* automatically optimized.) Thirdly, one can exploit data parallelism for map-like recursive functions by transforming them into task farms.

For this third reason, it is worth to capacitate pattern candidate discovery to find map-like functions. Here we propose a set of conditions to be used for the identification of map-like functions. What are the main characteristics of a map-like function f? It should have a list parameter, and return a list. The head of the returned list may depend on the head of the input list, but may not depend on its tail. Similarly, the tail of the returned list may not depend on the head of the input list: it should simply be the result of a recursive call to f on the tail of the input list. The function may have additional parameters, but all these parameters must be passed to the recursive call unchanged.

```
1  f(Parameter,List) ->
2    case List of [] -> [];
3                 [Head|Tail] -> X = ... Head ...,
4                                [X | f(Parameter,Tail)]
5    end.
```

There are many syntactic forms that satisfy these requirements, one is shown above. Note that f need not call itself directly, we allow mutually recursive functions as well.

Now let us investigate these requirements in a more formal way. We propose conservative rules to use for identifying map-like functions. Consider a function f. We will rely on the control flow and data flow graphs presented in section 3 built using f as the entry point. Thanks to the interprocedural analyses, mutually recursive functions are also handled properly.

1. f must be recursive: the interprocedural CFG must contain a directed path from the starting node of f to a *call*-node of f.
$$\exists p \in EP(start_f) \colon call_f \in p$$
2. The definition of f must have a base case: there is a directed path from the starting node of f to the ending node of f which does not include a *call*-node of f.
$$\exists p \in EP(start_f) \colon \big(call_f \notin p\big) \wedge \big(end_f \in p\big)$$
3. f should not be of divide-and-conquer style (like a quicksort), in the sense that it should not recurse multiple times: on each execution path, after a recursive call to f has returned, no new call to f may occur.
$$\forall p \in EP(ret_f) \colon call_f \notin p$$

4. f may have multiple parameters, among which there is one, a list, which will be processed element-by-element. Therefore, we assume that the definition of f is provided with a single clause, the head of which is $f(\overline{V_1}, \mathtt{L}, \overline{V_2})$, where parameter `L` corresponds to the list consumed by f. (If f is defined with multiple clauses, we can meld those into a single clause by introducing an artificial `case`-expression that contains the pattern matching and the guards of the original function clauses.) We can define the semantic functions $Head(\mathtt{L})$ and $Tail(\mathtt{L})$ based on the data-flow graph as follows.

$$Head(\mathtt{L}) = \left\{ n \in N \;\middle|\; \exists n' \in N : \mathtt{L} \overset{\mathbf{0f}}{\rightsquigarrow} n',\ n' \overset{\mathbf{s_h}}{\rightarrow} n \right\}$$
$$Tail(\mathtt{L}) = \left\{ n \in N \;\middle|\; \exists n' \in N : \mathtt{L} \overset{\mathbf{0f}}{\rightsquigarrow} n',\ n' \overset{\mathbf{s_t}}{\rightarrow} n \right\}$$

5. The expression returned by f must satisfy further properties. Regarding the different execution paths of f, we can define the set R of return expressions by collecting the last evaluated expression on all paths before reaching the ending node of f. The last evaluated expression is connected to the ending node by an edge in the CFG.

$$R = \left\{ \varrho \;\middle|\; (\varrho, l, end_f) \in E \text{ (for some label } l) \right\}$$

For all return expression $\varrho \in R$, the following requirements must hold.

(a) If ϱ is on a non-recursive execution path, then ϱ must evaluate to an empty list, i.e. only an empty list constructor is in the compact data-flow reaching relation with ϱ.

$$\forall n \in N\colon \quad n \overset{\mathbf{0f_c}}{\rightsquigarrow} \varrho \implies n = \mathtt{[]}$$

(b) If ϱ is on a recursive path, then ϱ must evaluate to a non-empty list, i.e. in the compact data-flow reaching only expressions in the syntactic form `[`χ`|`τ`]` or `[`χ`]` `++` τ (where χ and τ are arbitrary expressions).

$$\forall n \in N\colon \quad n \overset{\mathbf{0f_c}}{\rightsquigarrow} \varrho \implies \Big(\exists \chi, \tau\colon n = \mathtt{[}\chi\mathtt{|}\tau\mathtt{]}\Big) \vee \Big(\exists \sigma, \tau\colon (n = \sigma \mathtt{++} \tau) \wedge \big(\forall n' \in N\colon n' \overset{\mathbf{0f_c}}{\rightsquigarrow} \sigma \implies \exists \chi\colon n' = \mathtt{[}\chi\mathtt{]}\big)\Big)$$

(c) With respect to the above condition, we set out further requirements on all possible χ and τ.

i. χ depends on `L` only through $Head(\mathtt{L})$, namely: for each $\mathtt{L} \overset{\mathbf{d}}{\rightsquigarrow} \chi$ path in the dependence graph, there exists $n \in Head(\mathtt{L})$ such that the path is a concatenation of paths $\mathtt{L} \overset{\mathbf{d}}{\rightsquigarrow} n$ and $n \overset{\mathbf{d}}{\rightsquigarrow} \chi$.

ii. τ is the result of the recursive call of f, where the parameters of the call are $\overline{V_1}$, the tail of `L`, and $\overline{V_2}$, namely $\exists n \in N, \forall n' \in N$:

$$n = f(\overline{\nu_1}, \alpha, \overline{\nu_2}) \ \wedge\ \overline{V_i} \overset{\mathbf{0f}}{\rightsquigarrow} \overline{\nu_i}\ (i = 1, 2) \ \wedge$$
$$n' \in Tail(\mathtt{L}) \ \wedge\ n' \overset{\mathbf{0f}}{\rightsquigarrow} \alpha \ \wedge\ n \overset{\mathbf{0f}}{\rightsquigarrow} \tau.$$

There are many ways to further generalize the analysis, in order to find more map-like functions. But now let's turn our attention to automated transformations. Once the above analysis identified a map-like function, we can try to automatically transform its body into a list comprehension. The following transformation rule is only applicable on a function that calls itself directly.

$$\frac{F(\overline{V_1}, L, \overline{V_2}) \to \overline{E}.}{\begin{array}{c} F(\overline{V_1}, L, \overline{V_2}) \to \textbf{\textit{Fun}} = \textbf{fun}\ (\textbf{\textit{H}}) \to \overline{E'}\ \textbf{end}, \\ [\ \textbf{\textit{Fun}}(\textbf{\textit{X}})\ ||\ \textbf{\textit{X}} \leftarrow L\ ]. \end{array}} \quad \text{WHEN } \begin{array}{l} \text{map\_like}(F), \\ \overline{E'} = \overline{E} \end{array}$$

$$\text{THEN IN } \overline{E'}\text{:} \quad \frac{L'}{[\textbf{\textit{H}}|ok]} \quad \text{WHEN } L \overset{\textbf{of}}{\rightsquigarrow} L'$$

$$\text{THEN IN } \overline{E'}\text{:} \quad \frac{Xs}{ok} \quad \text{WHEN } Xs \in \mathit{Tail}(L)$$

$$\text{THEN IN } \overline{E'}\text{:} \quad \frac{F(\overline{V_1'},\ L',\ \overline{V_2'})}{ok}$$

$$\text{THEN IN } \overline{E'}\text{:} \quad \frac{X}{\textbf{\textit{H}}} \quad \text{WHEN } X \in \mathit{Head}(L)$$

$$\text{THEN IN } \overline{E'}\text{:} \quad \frac{Z}{hd(Z)} \quad \text{WHEN } Z \in R$$

We define the kernel function of the computation as a lambda expression, and replace the body of the function with this definition and a list comprehension. We generate the kernel function (`fun(H)->...end`) by substituting the occurrences of the list L with an improper list `[H|ok]`, the occurrences of the tail of the list L and the recursive calls with the atom `ok`, the occurrences of the head of the input list with the variable `H`, and we apply the head (`hd/0`) function over the element of the return points of the function. Cleanup transformation should be performed to simplify the resulted kernel.

5 Related Work

Several approaches have been proposed either to aid the identification of code fragments that are amenable to parallelization, or to provide patterns and refactorings to introduce (structured) parallelism. Also, many papers report on significant effort put into automatic, compile-time optimisation that yields parallel object code. Most of these projects focused on mainstream languages (mostly imperative and object-oriented). In Erlang, recurring computations implemented by (first-order and higher-order) recursive functions are of most interest, but

in the imperative setting, automatic loop parallelization is in the very focus of related research.

The approach proposed by Hammacher *et al.* [9] focuses on parallelization of sequential, legacy software written in Java. By considering dependencies that are determined by using dynamic dependency graphs, they identify independent program paths that can be run in parallel. They can make suggestions on where to apply parallelism, but do not provide support on actual code shaping.

There has also been some work on using static analysis to discover parallelism bottlenecks, and providing help to the programmer to reshape the program in accordance with such analysis. The results presented in [10,11,12] use static analyses to i) introduce parallelism in X10 programs by refactoring, and ii) discover parallelism bottlenecks and provide thread safety and/or reentrancy. Compared to our approach, they do not consider patterns as instances of algorithmic skeletons, nor provide suggestions on which skeletons to apply.

Molitorisz [13] describes a tool for automatic discovery of parts of a sequential application that implement some rather basic parallelization patterns, as well as proposes supporting refactoring and performance tuning techniques. The same author presents AutoFutures [14], a tool that performs static analysis on Java programs to discover portions of code showing no data dependencies, and inserts parallel constructs (Futures) directing asynchronously parallel execution. Unlike the work presented in this paper, however, the author does not take into account general algorithmic skeletons, and moreover does not present possible pattern candidates to the user for further program shaping and parallel refactoring. Work by Dig [15,16,17] introduces concurrency in Java programs, also by targeting thread safety, aiming to increase throughput and scalability. Dig's refactoring tool contains a minor selection of transformations that are able to rewrite Java code so that it employs generic Java Concurrency libraries to control parallel execution. In [18], the authors report on promising results in automatic shaping and parallelization in C code based on dependence profiling.

Many consider structured parallelism as the answer to various issues related to parallelization of large-scale systems. The skeletons research community has been working on methods for parallel programming in high-level languages since the nineties [5,19,20]. This has resulted in a number of skeletons/patterns and skeleton libraries in a range of languages including C, C++ and Java. Automatic parallelization in this context means rewriting sequential code into instances of parallel skeletons. In [21], Brown *et al.* have proposed a small number of basic refactorings to introduce farm and pipeline skeletons into C++ using the FastFlow library.

In the context of functional programming languages, fewer approaches have been proposed, both in static analysis based pattern detection or automatic shaping. Earlier attempts at parallelizing Erlang programs are ad hoc approaches that can be found in the parallelization of Dialyzer [22] and a suite of Erlang benchmarks [23]. None of these applied structured parallelism or parallel skeletons.

For Haskell, Brown *et al.* introduced a limited number of parallel refactorings [24] in *HaRe* – the Haskell Refactorer. This work introduces parallelism

using structural refactorings. Finally, a skeleton-based parallel compiler for ML – SkelML [25] – was introduced that automatically identifies certain forms of parallel skeleton. SkelML does not present pattern candidates to the programmer or allow program shaping and parallel refactorings, thus giving the programmer choice and guidance into which parallelization to perform.

6 Conclusion

Expressing parallelism with algorithmic skeletons can significantly reduce the effort needed to develop effective and reliable parallel programs. Selecting sequential code fragments and transforming them into parallel is not straightforward in the case of large-scale software, therefore some kind of methodology and tool support is needed. The ParaPhrase Refactoring Tool for Erlang (PaRTE) provides features like pattern discovery, profiling, ranking and semi-automatic transformations. This paper described the static program analyses used by the pattern discovery in PaRTE, as well as the formal rules of (some of) the program shaping transformations offered by this refactoring tool. The transformations explained here complement earlier work [2,6] and facilitate the automatic parallelization of many syntactic forms.

References

1. Hammond, K., Aldinucci, M., Brown, C., Cesarini, F., Danelutto, M., González-Vélez, H., Kilpatrick, P., Keller, R., Rossbory, M., Shainer, G.: The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In: Formal Methods for Components and Objects. Volume 7542 of LNCS. Springer Berlin Heidelberg (2013) 218–236
2. Bozó, I., Fördős, V., Horváth, Z., Tóth, M., Horpácsi, D., Kozsik, T., Kőszegi, J., Barwell, A., Brown, C., Hammond, K.: Discovering parallel pattern candidates in erlang. In: Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang. Erlang '14, New York, NY, USA, ACM (2014) 13–23
3. Skel Tutorial. Available at http://chrisb.host.cs.st-andrews.ac.uk/skel-test-master/tutorial/bin/tutorial.html (2014)
4. The ParaPhrase project. http://www.paraphrase-ict.eu (2014)
5. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, MA, USA (1991)
6. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliot, A.: Cost-Directed Refactoring for Parallel Erlang Programs. Int'l J. of Parallel Programming (2013)
7. Brown, C., Janjic, V., Hammond, K., Goli, M., McCall, J.: Bridging the Divide: Intelligent Mapping for the Heterogeneous Parallel Programmer. ICPP'13 (2013)
8. Tóth, M., Bozó, I.: Static Analysis of Complex Software Systems Implemented in Erlang. In: Central European Functional Programming School. Volume 7241 of Lecture Notes in Computer Science. Springer (2012) 440–498
9. Hammacher, C., Streit, K., Hack, S., Zeller, A.: Profiling Java Programs for Parallelism. In: Proc. IWMSE '09. (2009) 49–55
10. Markstrum, S.A., Fuhrer, R.M.: Extracting Concurrency via Refactoring in X10. In: Proceedings of the 3rd ACM Workshop on Refactoring Tools. WRT'09 (2009)

11. Markstrum, S.A., Fuhrer, R.M., Millstein, T.D.: Towards Concurrency Refactoring for x10. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '09, New York, NY, USA, ACM (2009) 303–304
12. Wloka, J., Sridharan, M., Tip, F.: Refactoring for Reentrancy. In: ESEC/FSE '09, Amsterdam, ACM (2009) 173–182
13. Molitorisz, K.: Pattern-Based Refactoring Process of Sequential Source Code. In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on. (March 2013) 357–360
14. Molitorisz, K., Schimmel, J., Otto, F.: Automatic Parallelization Using Autofutures. In: Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools. MSEPT'12, Berlin, Heidelberg, Springer-Verlag (2012) 78–81
15. Dig, D.: A Refactoring Approach to Parallelism. IEEE Softw. **28** (2011) 17–22
16. Dig, D., Marrero, J., Ernst, M.D.: How do programs become more concurrent: A story of program transformations. In: Proceedings of the 4th International Workshop on Multicore Software Engineering. IWMSE '11, New York, NY, USA, ACM (2011) 43–50
17. Dig, D., Marrero, J., Ernst, M.D.: Refactoring sequential java code for concurrency via concurrent libraries. In: Proceedings of the 31st International Conference on Software Engineering. ICSE '09, Washington, DC, USA, IEEE Computer Society (2009) 397–407
18. Mak, J., Faxén, K.F., Janson, S., Mycroft, A.: Estimating and exploiting potential parallelism by source-level dependence profiling. In: Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I. EuroPar'10, Berlin, Heidelberg, Springer-Verlag (2010) 26–37
19. Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Comput. **30**(3) (March 2004) 389–406
20. Darlington, J., Guo, Y., Jing, Y., To, H.W.: Skeletons for Structured Parallel Composition. In: Proc. of the 15th Symposium on Principles and Practice of Parallel Programming. (1995)
21. Brown, C., Janjic, V., Hammond, K., Schöner, H., Idrees, K., Glass, C.W.: Agricultural Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools. In: Proc. PDP '14, IEEE (2014)
22. Aronis, S., Sagonas, K.: On Using Erlang for Parallelization : Experience from Parallelizing Dialyzer. Proceedings of the Symposium on Trends in Functional Programming (2012)
23. Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., Venetis, I.E.: A Scalability Benchmark Suite for Erlang/OTP. In: Proc. of 11th ACM SIGPLAN workshop on Erlang, New York, NY, USA, ACM (2012) 33–42
24. Brown, C., Loidl, H., Hammond, K.: Paraforming: Forming Haskell Programs using Novel Refactoring Techniques. In: Twelth Symposium on Trends in Functional Programming, Madrid, Spain (May 2011)
25. Michaelson, G., Ireland, A., King, P.: Towards a Skeleton Based Parallelising Compiler for SML. In: Proceedings of 9th International Workshop on Implementation of Functional Languages. (1997) 539–546

Discovering Parallel Pattern Candidates in Erlang

István Bozó Viktória Fördős Dániel Horpácsi Adam Barwell Chris Brown
Zoltán Horváth Melinda Tóth Tamás Kozsik Judit Kőszegi Kevin Hammond

ELTE-Soft Nonprofit Ltd Eötvös Loránd University University of St Andrews
{bozoistvan,f-viktoria,hz,tothmelinda}@elte.hu {daniel-h,kto,koszegijudit}@elte.hu {adb23,cmb21,kh8}@st-andrews.ac.uk

Abstract

The *ParaPhrase Refactoring Tool for Erlang* (**PaRTE**) provides automatic, comprehensive and reliable pattern candidate discovery to locate parallelisable components in Erlang programs. It uses semi-automatic and semantics-preserving program transformations to reshape source code and to introduce high level parallel patterns that can be mapped adaptively to the available hardware resources. This paper describes the main **PaRTE** tools and demonstrates that significant parallel speedups can be obtained.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Languages; Design; Performance

Keywords Refactoring; Erlang; Skeletons; Patterns; ParaPhrase; Parallelism; Concurrency

1. Introduction

Pattern candidates are code fragments that can be transformed into instances of well-known high-level parallel patterns. The EU **ParaPhrase** project [18] aims to produce new tools and techiques that can: i) (semi-)automatically locate suitable pattern candidates in Erlang programs; and ii) propose transformed versions of these pattern candidates that yield significant speedup on a given parallel architecture. The *ParaPhrase Refactoring Tool for Erlang* (**PaRTE**) identifies pattern candidates in Erlang programs, using static analyses to determine:

- which operations are performed element-wise on multiple (perhaps many) data items;
- what type of data these operations are applied to; and
- whether these operations may be executed simultaneously (in parallel) without changing the outcome of the computation.

These analyses are used to select transformable structures, to verify side conditions and to prioritise pattern candidates. The *Pattern Candidate Browser* reports pattern candidates and recommends good parallelisations, presenting information in a way that avoids overloading its users with unnecessary details, but which highlights the key decisions that must be made. This paper describes how **PaRTE** can be used to parallelize programs, focusing on how parallel pattern candidates are selected by the *Pattern Candidate Browser*, and on the application of the corresponding transformation sequence by the underlying refactoring engine. The main novel contributions of the paper are:

1. we introduce a new automatic *pattern discovery* tool and technique that can discover instances of algorithmic skeletons within sequential Erlang programs;
2. we describe a new *pattern candidate browser* that suggests possible parallelisations and provides speedup predictions;
3. we introduce a number of new semi-automatic (user-driven) *program shaping* refactorings, that facilitate the re-structuring of Erlang programs so that the required parallelisation can then be introduced (also via refactoring);
4. we introduce a new inter-operable framework, marrying pattern discovery, refactoring techniques and algorithmic skeletons to provide the Erlang programmer with tool set for parallel programming; and,
5. we demonstrate the use of our approach on an image merging application, showing speedups of around 13 on a 24-core shared memory system.

2. Parallel Patterns and Algorithmic skeletons

Whilst Erlang's concurrency model provides relatively simple primitives, the programmer must still avoid deadlocks, create and manage processes, etc. The need to explicitly consider such low-level details means that parallelisation remains difficult, tedious, and error-prone for the average programmer. As programs grow ever larger and run on hardware with increasingly numerous cores, it is increasingly important to tackle this problem directly. Parallel design patterns offer one promising approach to achieve this [26]. *Algorithmic skeletons* [11] implement common patterns of parallelism, allowing the programmer to instantiate parallel skeletons with application-specific code fragments. They are often implemented as higher-order functions [12]. *Skel*[1] is a new library of algorithmic skeletons for Erlang, providing a small number of useful, classical skeletons [9]. These include, but are not limited to, the following skeletons.

- `pipe` is a parallel pipeline, applying each of its stages in turn to a sequence of independent inputs. The output of any one stage acts as input for the next, until the end of the pipeline is reached.
- `farm` applies a given function to a sequence of independent inputs in parallel.
- `seq` is a trivial wrapper skeleton that allows sequential code fragments to be embedded within other skeletons.

[1] http://skel.weebly.com

Erlang '14, September 5, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3038-1/14/09. . . $15.00.
http://dx.doi.org/10.1145/2633448.2633453

To illustrate their use, we will consider a typical application: an image merging operation. A single application of image merging takes a pair of images, producing a single combined image. This process may be split into two stages: i) reading the images into memory; and ii) performing the actual merge. The image merge can be applied to a list of image pairs, using e.g. a *map*.

```
merge(Images) ->
    lists:map(fun(Y) -> convertMerge(readImage(Y)) end
       , Images).
```

The larger and more numerous the images, the longer the image merge process will take. Parallelisation can therefore lead to major performance gains. This is similarly true of any operation applied to every object in a stream of independent inputs.

The *merge* operation could obviously be parallelised directly using Erlang primitives. One approach is to spawn a process for each image pair and stage. To implement this, a collector process needs to be created to collect merged images. The `pid` of this collector is used to spawn another process that creates processes to merge each image pair. The `pid` of the merger generator is in turn used to load each image pair into memory. This is done by spawning a reading process for each pair. Once both images in a pair are loaded, they are passed to the merger generator, and the reading process discarded. The merger generator works in a similar manner, it spawns a merging process for each pair of images it receives. This merging process performs the merge operation, and passes the result to the collector. Once the collector receives all merged images, it then passes the results back to our originally called function, which returns with that result. Approximately forty lines of code is required to implement this system. Alternatively, we can parallelise the image merge operation using *Skel*, as shown below.

```
merge(Images) ->
  skel:do([{farm,
            [{pipe,
              [{seq, fun ?MODULE:readImage/1},
               {seq, fun ?MODULE:convertMerge/1}]}],
            NW}], Images).
```

Here, only one *Skel* invocation is needed. This takes a list of skeletons (here, a task farm containing a two-stage pipeline) plus the list of image pairs to be merged, `Images`. The library performs all the necessary parallel process generation and manipulation. The programmer only needs to provide the problem-specific code fragments and decide which skeletons to use. Unlike the conventional manual approach, there is no need to consider how best to create processes, nor how to marshal and collect individual image pairs. This increases code abstraction and readability, reduces opportunities for errors, and makes it easier to alter code structure and/or parallel implementations in future. We will explore this example in greater depth in Section 6.

3. The PaRTE framework

PaRTE integrates capabilities of the *RefactorErl* [6, 32, 34] and *Wrangler* [22] refactoring/program analysis tools into a new parallelisation framework that can be used to identify parallel patterns and determine the best implementations of those patterns. Both the pattern candidate detection/assessment and semantics-preserving transformation steps require thorough syntactic and semantic analysis. To do this, we exploit the *RefactorErl* and *Wrangler* refactoring tools, which implement specific compile-time analyses, and support syntax-based transformations. *RefactorErl* implements a wide range of static semantic analyses, including scope analysis of various language entities, side-effect analysis and callee approximation of dynamic function calls. Most of these (higher-level) semantic analyses build on results from dataflow and type analyses. Based on the information uncovered by the semantic analyses, *RefactorErl* can determine non-trivial properties of code fragments. In the integrated **PaRTE** framework, it is therefore used to perform pattern discovery and evaluation. Conversely, *Wrangler* has a mature user interface for performing simple code rewriting, which makes it a good choice for a tool to carry out parallelisation transformations. In the integrated **PaRTE** framework, *Wrangler* is responsible for defining and executing shaping transformations, and for turning sequential computations into instances of parallel algorithmic skeletons.

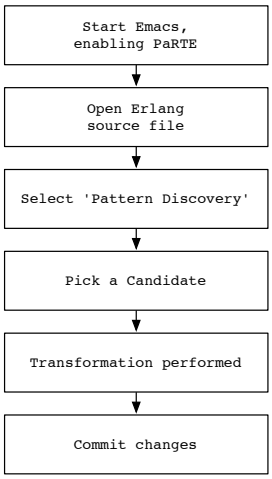


Figure 1. Stages of using **PaRTE**

3.1 Using PaRTE

This section describes how to use the **PaRTE** tools, using Emacs to provide the user interface (in principle, other IDEs could be used, if preferred). The integration of the different tools and interfaces is seamless to the programmer, the installation as well as the execution of the tool chain is managed and supervised by **PaRTE**. Here we describe the approach taken by a single user on local Erlang source files (Figure 1). Since some of the **PaRTE** tools already allow multiple simultaneous users, it is reasonable to extend this to support collaborative development in the future.

The Erlang developer starts Emacs, and enables the PaRTE mode. This will start **PaRTE**. In more detail, the following will happen. Firstly, an Erlang node, called *wrangler* will be started: *Emacs* will communicate with this node. The *wrangler* node will start the *Wrangler* application, as well as another Erlang node, *refactorerl*, which will start the *RefactorErl* application, and introduce a server that is responsible for communication between *Wrangler* and *RefactorErl*.

The developer opens an Erlang source file in Emacs. *Emacs* notifies the *wrangler* node, which in turn notifies the *refactorerl* node. This loads the contents of the file into the *RefactorErl* database.

The developer selects the* Pattern discovery *menu item. Our tool analyses the freshly loaded code, collecting pattern candidates as described in Section 4. Candidates satisfying the transformation side-conditions will then be presented in the *Pattern Candidate Browser* (Section 4.4) together with parallel execution time estimates and speedup predictions.

A candidate is chosen by the developer in the Pattern Candidate Browser. A pattern candidate specifies how a given code frag-

ment can be transformed into a skeleton structure. For instance, a pattern candidate might specify that a *farm* of *pipes* should be introduced. The necessary transformation (which is often quite complex) is expressed as a sequence of smaller transformations, which include *shaping transformations* (Section 5) plus transformations to introduce *seq*, *pipe*, *farm* etc. skeletons.

Transformations are performed under developer supervision. Having selected a pattern candidate, **PaRTE** will carry out the required transformation sequence. A preview will be shown before major changes, offering the developer the possibility of cancelling the transformations and revert to the original state. Section 5.5 elaborates on the technical details of this step.

The developer commits the changes. Once the entire transformation sequence is completed, the developer may commit the changes to the source code. At this point, *Emacs* instructs the *wrangler* node to save the modified source code and instructs the *refactorerl* node to reload and reanalyse these files.

4. Pattern Candidate Discovery

We aim to automatically discover code fragments that match high-level parallel patterns. Since we are targeting the skeletons from the *Skel* library, in particular *farm* and *pipe*, which are defined to operate on lists of inputs, our analysis focuses on identifying certain operations on lists, and also on identifying those data structures that can be transformed to lists. We target three constructs: list comprehensions, library calls and recursive functions.

Analyzing list comprehensions. Pattern discovery categorises every list comprehension as a possible *farm* or *pipe* pattern based on its head expression. For example, if the head expression syntactically contains a function call, then it may be computationally intensive, and hence we can categorize it as a farm candidate, or if the head expression composes two or more functions, then it can be categorized as a pipe candidate. Likewise, if the head expression calls a function, which calls another function in its body, then we can also categorize it as a possible pipe candidate.

Analyzing library calls. We have examined some commonly-used Erlang library modules, and identified those library functions that exhibit a map-like or pipeline-like behaviour over a sequence of data. Calls to these functions can be transformed into e.g. *farms* or *pipes*. We initially focus on the list-manipulating modules `lists`, `proplists`, `ordsets` and `orddict`. We also consider functions from `array`, `sets` and `dict`, since they manipulate data that can be transformed to lists. Finally, we consider the `ets`, `dets` and `mnesia` modules, which are used to manipulate collections of values. Since many of these operations in these modules have side-effects, however, their parallelisation is not straightforward. Calls to such functions therefore need to consider the calling context to determine whether they can be used safely: side-condition analysis (Section 4.2) is used to check this.

Identifying map-like and pipeline-like recursive functions. We can identify two kinds of recursive definitions as pattern candidates: map-like and pipeline-like functions. Pattern discovery analyses function definitions and searches for certain program structures. In order to identify a given recursion scheme, we need to combine syntactic and semantic (e.g. dataflow and type) information. As before, we only consider functions that recurse on the elements of a list. All the candidates that are found by the above analyses are further examined, and certain side-conditions are checked for parallel safety as described in Section 4.2.

For example, consider the analysis for identifying map-like functions given a function definition of the following syntactic form (for simplicity, we will assume that the formal parameters are variables, and that function clauses are replaced by `case` expressions):

$$f(\overline{V_1}, L, \overline{V_2}) \rightarrow \varepsilon^+$$

The body of the function is a non-empty sequence of ε expressions. The function must statically (and conservatively) satisfy the following type requirement: *(any()*, list(), any()*) → list()*. The possible execution paths of this function can be calculated using an inter-procedural control flow analysis. For each execution path, either of the following two cases must hold.

1. Function f is called recursively exactly once on the path (we assume both static and dynamic call analysis here). Furthermore, if the return value produced by this execution path is ϱ, the following two requirements are fulfilled:

$$\begin{aligned} \mathrm{Head}(\varrho) &\equiv \varphi(\overline{V}_1, \mathrm{Head}(L), \overline{V}_2) \\ \mathrm{Tail}(\varrho) &\equiv f(\overline{V}_1, \mathrm{Tail}(L), \overline{V}_2) \end{aligned}$$

   where Head and Tail are semantic functions that take the head and the tail of a list, respectively, and φ is some function depending only on $\mathrm{Head}(L)$, $\overline{V}_1$ and $\overline{V}_2$. The symbol $\equiv$ is semantic equivalence; we apply data-flow analysis to compute the possible values of expressions.

2. Function f is not called recursively at all on the execution path. In this case, $L \equiv []$, and for the return value ϱ of this path: $\varrho \equiv []$.

Moreover, all the expressions in f that are not on the path containing the recursive call of f must depend on L only through $\mathrm{Head}(L)$. This latter requirement is verified by dataflow analysis.

4.1 An example: matrix multiplication

We illustrate the process of pattern discovery using a simple matrix multiplication example. The `mult_matrix/2` function takes two matrices (the first of which is represented as a list of rows, and the second as a list of columns), and calculates their product.

```
mult_matrix(Rows, Cols) ->
  [scalar_product(R,C) || R <- Rows, C <- Cols].

scalar_product(R, C) ->
  lists:sum([mult_scalar(A,B) ||
             {A,B} <- lists:zip(R,C)]).

mult_scalar(A, B) ->
  A*B.
```

In order to show pattern discovery on this example, we have deliberately introduced the unnecessary `mult_scalar` function. This will trick the analysis into believing that multiplying two scalars is computationally intensive, which will allow it to identify more pattern candidates. Pattern discovery will identify both of the list comprehensions as possible parallel *farms*. Furthermore, the first list comprehension is also identified as a possible parallel *pipe*. Based on the pattern candidates that have been discovered and information about the control flow of the program, we may be able to construct *composite* pattern candidates from these base pattern candidates, for example, for the matrix multiplication example, we could nest a *farm* within a *pipe*, as shown below.

```
mult_matrix(Rows, Cols) ->
  [scalar_product(R,C) || R <- Rows, C <- Cols].

scalar_product(R, C) ->
  skel:do([{pipe, [{farm, [{seq, fun(A,B) ->
                              mult_scalar(A,B) end}],
                           NW},
                   {seq, fun lists:sum/1}]}]
                   , lists:zip(R,C)).
```

```
mult_scalar(A, B) ->
  A*B.
```

Here, we introduce a two-stage *pipeline*. The first stage of the pipeline is a *farm* skeleton, where the worker is a sequential wrapper (denoted by `seq`) to the `lists:sum/1` function and `NW` corresponds to the number of workers (a user-defined value). In the second stage of the pipeline, we have a sequential wrapper to the `mult_scalar` function. An alternative definition of the same example can be given using a recursive function and a call to the `lists:map/2` function.

```
mult_matrix2([], _) ->
  [];
mult_matrix2([R | Rows], Cols) ->
  [lists:map(fun(C) ->
               scalar_product(R,C)
             end, Cols) | mult_matrix2(Rows,Cols)].
```

In this case, pattern discovery will identify the applications of the `lists:map/2` and `mult_matrix2/2` functions as *farm* candidates. The resulting parallel implementations would be e.g.

```
mult_matrix2([], _) ->
  [];
mult_matrix2([R | Rows], Cols) ->
  [ skel:do([{ farm,
               [{seq,
                 fun(C) -> scalar_product(R,C) end
               }],
               NW }], Cols)
  | mult_matrix2(Rows,Cols)].
```

Here, we introduce a *farm* inside the recursive call for the call to the `scalar_product` function. Alternatively, by introducing parallelism at the outer level and removing the recursion, we would obtain:

```
mult_matrix2(Rows, Cols) ->
  skel:do([{ farm
           , [{seq,
               fun(R) ->
                  lists:map(fun(C) ->
                               scalar_product(R,C) end,
                            Cols)
               end}]
           , NW }], Rows).
```

Skeletons are introduced using *Wrangler* refactorings that provide calls to the *farm* and *pipeline* skeletons [9] semi-automatically (i.e., under programmer control). These refactorings are described in more detail in Section 5.1.

4.2 Side-conditions and Components

Fundamentally, pattern discovery is syntax-driven. However, not all code fragments that match a pattern can be safely executed in parallel. Some additional preconditions must also be met in order to guarantee that the refactoring transformations preserve the dynamic semantics of the program. These checks are termed *side-condition analysis*. They are realised by building on static semantic analyses implemented in the refactoring tools, extending and annotating the syntax tree with a wide range of context-dependent relations between program entities. Such semantic analyses include, for instance, call-graph analysis, data- and control-flow analyses and side-effect analysis. Side-effect analysis plays an especially important role in the side-condition analysis process, because i) it is necessary to determine whether a computation has side-effects when computations are overlapped and/or reordered; and ii) nontrivial code analysis (and representation) is needed in order to determine purity properties. Since it is almost impossible to find completely pure legacy Erlang code and absolute purity may not, in fact, be necessary to ensure safe parallel execution, in our approach, components are permitted to have some carefully defined combinations of side-effects, under which they are still considered to give sensible results when executed in parallel. If all the components of a pattern candidate are sufficiently well-behaving according to our definition, then the parallelisation of that candidate is considered to be safe.

Pattern Components and Side-Effect Analysis. Erlang expressions may refer to global state in various ways (e.g. through the language run-time environment, such as node names and process dictionary; through global data sources, such as Erlang Term Storage (ETS); through message passing etc.), hence they may read or alter global resources.

$$\varepsilon \uparrow \rho \iff \text{expression } \varepsilon \text{ alters resource } \rho$$
$$\varepsilon \downarrow \rho \iff \text{expression } \varepsilon \text{ reads resource } \rho$$

We collect sets of read and altered resources using dataflow and process analyses, and attach them to expressions. Each expression will therefore have both a "read resources" set and an "altered resources" set, defined by the images of the above relations. When the dataflow analysis cannot determine the resources that are read or altered by an expression, the corresponding set will contain a special "unknown resource" value, denoted by $\perp$.

Our side-effect analysis distinguishes three kinds of expressions: white, grey and black. Grey means that the expression reads some global state; black means that the expression alters some global state; white expressions neither read nor alter the global state.

$$\begin{aligned} \mathit{black}(\varepsilon) &\iff (\exists \rho : \varepsilon \uparrow \rho) \\ \mathit{grey}(\varepsilon) &\iff (\exists \rho : \varepsilon \downarrow \rho) \ \wedge (\nexists \rho : \varepsilon \uparrow \rho) \\ \mathit{white}(\varepsilon) &\iff (\nexists \rho : (\varepsilon \downarrow \rho \vee \varepsilon \uparrow \rho)) \end{aligned}$$

Exceptions may also be considered to be a source of impurity [31]. For our purposes, however, it is reasonable to consider expressions that can potentially raise exceptions as white.

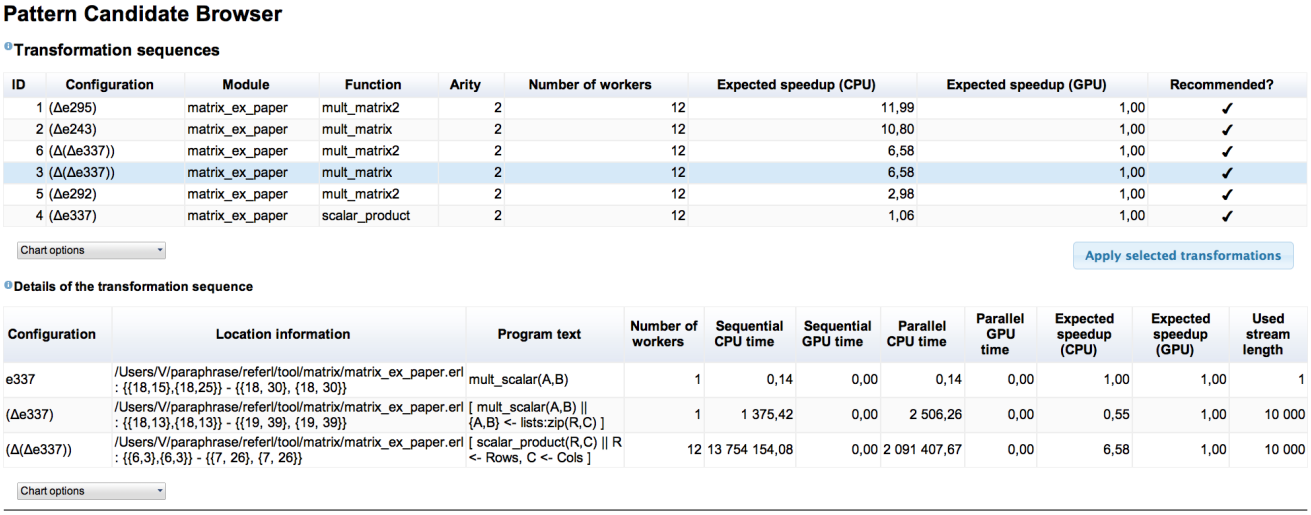
A component set $\mathcal{C}$ is defined by its elements, while a component C is given as a series of its expressions. The side-effects of the component are simply inherited from its expressions.

$$\mathcal{C} = \{C_1 \ldots C_l\}$$
$$C = \ll \varepsilon_1 \ldots \varepsilon_k \gg$$
$$C \uparrow \rho \iff \exists i : \varepsilon_i \uparrow \rho \quad \text{(component } C \text{ alters resource } \rho\text{)}$$
$$C \downarrow \rho \iff \exists i : \varepsilon_i \downarrow \rho \quad \text{(component } C \text{ reads resource } \rho\text{)}$$

The rules governing the purity of components can be stated as follows. Consider a set of component candidates (i.e. a set of program units that will run in parallel within a skeleton). If all components in this set contain only white and grey expressions, then the set will be considered to be sufficiently well-behaved. If one of the components alters a specific resource, that resource cannot be read or altered by expressions in any other component. If a component has an "unknown resource" in its black set, all other components in the set may only contain white expressions. If a component contains an "unknown resource" in its grey set, no other component in the set is allowed to alter any resource.

$$\begin{aligned} \mathit{well\text{-}behaved}(\mathcal{C}) &\iff \nexists (i,j) : \big(i \neq j \wedge \mathit{clash}(C_i, C_j)\big) \\ \mathit{clash}(C_i, C_j) &\iff \exists \rho : \big((C_i \uparrow \rho \wedge (C_j \uparrow \rho \vee C_j \downarrow \rho)) \vee \\ &\qquad (C_i \uparrow \perp \wedge (C_j \uparrow \rho \vee C_j \downarrow \rho)) \vee \\ &\qquad (C_i \downarrow \perp \wedge C_j \uparrow \rho)\big) \end{aligned}$$

If a set of components is sufficiently well-behaved according to this definition, then **PaRTE** will allow the pattern candidate to be selected as a possible parallelisation.



Pattern Candidate Browser

Transformation sequences

ID	Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)	Expected speedup (GPU)	Recommended?
1	(Δe295)	matrix_ex_paper	mult_matrix2	2	12	11,99	1,00	✓
2	(Δe243)	matrix_ex_paper	mult_matrix	2	12	10,80	1,00	✓
6	(Δ(Δe337))	matrix_ex_paper	mult_matrix2	2	12	6,58	1,00	✓
3	(Δ(Δe337))	matrix_ex_paper	mult_matrix	2	12	6,58	1,00	✓
5	(Δe292)	matrix_ex_paper	mult_matrix2	2	12	2,98	1,00	✓
4	(Δe337)	matrix_ex_paper	scalar_product	2	12	1,06	1,00	✓

Chart options

Apply selected transformations

Details of the transformation sequence

Configuration	Location information	Program text	Number of workers	Sequential CPU time	Sequential GPU time	Parallel CPU time	Parallel GPU time	Expected speedup (CPU)	Expected speedup (GPU)	Used stream length
e337	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,15},{18,25}} - {{18, 30}, {18, 30}}	mult_scalar(A,B)	1	0,14	0,00	0,14	0,00	1,00	1,00	1
(Δe337)	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,13},{18,13}} - {{19, 39}, {19, 39}}	[mult_scalar(A,B) || {A,B} <- lists:zip(R,C)]	1	1 375,42	0,00	2 506,26	0,00	0,55	1,00	10 000
(Δ(Δe337))	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{6,3},{6,3}} - {{7, 26}, {7, 26}}	[scalar_product(R,C) || R <- Rows, C <- Cols]	12	13 754 154,08	0,00	2 091 407,67	0,00	6,58	1,00	10 000

Chart options

Figure 2. Pattern Candidate Browser

4.3 Exploiting Profiling Information

In a typical sequential Erlang program, pattern discovery may identify numerous valid pattern candidates. However, only a few of these could significantly improve program performance when executed in parallel. We use profiling to evaluate these pattern candidates and to produce an ordering with respect to the expected performance gain. This evaluation can then be presented to the developer, using e.g. the *Pattern Candidate Browser* discussed below, allowing the developer to make good decisions on the possible parallelisation actions. The possible performance improvement is determined by calculating the ratio of the (estimated) sequential and parallel execution time, to give an *estimated absolute speedup*. The execution time of each pattern candidate is estimated using pattern-specific cost models (described in earlier work, e.g. in [9]), which are parametrised on the execution time of the body of the computation. Profiling provides these cost models with information about the sequential execution time (or more generally, time complexity) of each component. In order to have a reasonably accurate estimation of the execution time of the component, we encapsulate the expression sequence that makes up the component into a function (lifting any free variables to function parameters), and invoke this function with a large number of randomly generated (well-typed) arguments. Function types are inferred by TypEr [23], and they are translated to QuickCheck [3] data generators controlling the distribution of test values. The average execution time that we obtain with these random values is taken to be the typical execution time of the component. Finally, we rank the candidates according to the values that we obtain by instantiating the cost models with these execution times, and present them to the developer using the *Pattern Candidate Browser*.

4.4 Pattern Candidate Browser

The *Pattern Candidate Browser* (Figure 2) presents pattern candidates to the developer and recommends the best transformation sequences. It is responsible for:

- displaying pattern discovery results in a user-friendly manner;
- forwarding the selected transformation sequence to be applied to the transformation system;
- storing pattern discovery data and making it available through a web service.

The task of the *Pattern Candidate Browser* is to provide a web-based user interface, where developers can browse and export pattern discovery results. It aims to present information in a way that avoids overloading developers with unnecessary details, but which highlights the key parallelisation decisions that must be made. Because of the client-server architecture of the interface, multiple developers (e.g., different members of a developer team) can browse pattern candidates simultaneously. Only one set of transformations can be carried out at a time, however. The flow of data between the different components of the *Pattern Candidate Browser* is shown in Figure 3. The two main server-side components are the following.

- **Persistence:** This produces the final representation of transformation sequences. Previously-stored transformation sequences can be queried by other components.
- **Web Server:** This serves content values via different protocols and handles user input.

Content values that are generated by one of the services are sent to the browser by the web server over HTTP. These may be HTML pages, XML or CSV files containing e.g. exported transformation sequences or transformation descriptions, JSON data or generated JavaScript scripts. This flexibility may, in the future, allow the Pattern Candidate Browser to be incorporated in the editor, or to constitute a separate tool.

Figure 2 shows how the *Pattern Candidate Browser* can be used on the matrix multiplication example. The upper table is an overview of the proposed transformation sequences. The lower table gives a more detailed view of a selected transformation sequence, with a description of each of its constituent transformation. A transformation sequence is characterized by a location (the entry point of the pattern candidate as a module-function-arity triple), an abstract skeletal configuration (which describes a composition of pipe, farm, etc. skeletons), and the expected speedup (for both CPU and GPU processors). The description of a transformation adds more specific location information, a view of the corresponding source code fragment, profiled sequential execution time on CPU and on GPU, and estimated parallel execution time on CPU and on GPU (GPU profiling is not currently implemented). The rows in the upper table are initially ordered by expected speedup (and hence implicitly by recommendation confidence), but re-ordering by another column is also possible. Developers can request that a

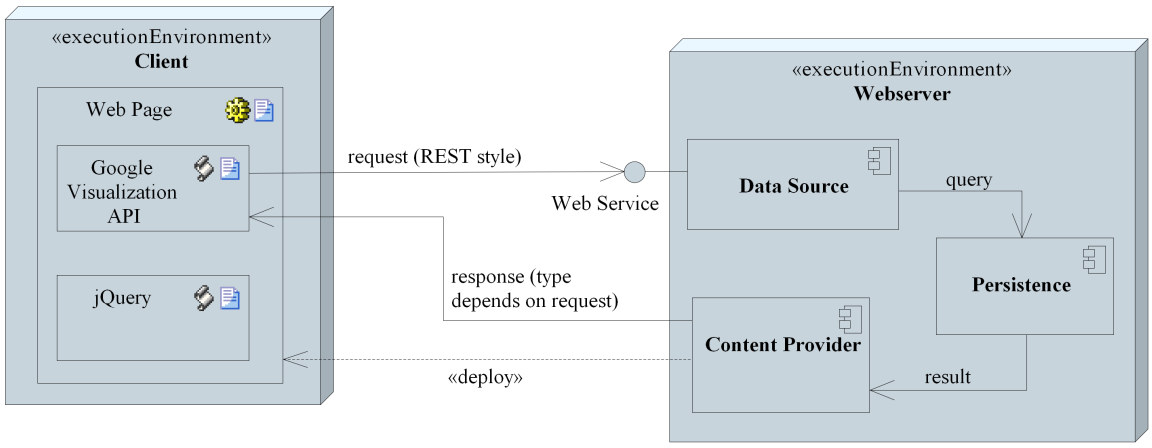


Figure 3. Interactions of the Pattern Candidate Browser Components

selected transformation sequence is applied by clicking on the *Apply selected transformations* button. Having confirmed the request, the *Pattern Candidate Browser* forwards this to the transformation system for action. This automatic transfer of control allows the developer to work conveniently in the editor (Emacs), but to browse pattern discovery results and select transformations to apply in a more appropriate tool.

5. Program Shaping and Parallel Refactoring

Once the developer has selected a pattern candidate, **PaRTE** automatically refactors the original code fragment into its parallel equivalent by introducing the appropriate parallel skeleton(s). The transformation process comprises two distinct phases: i) an initial *program shaping* phase; and ii) actually rewriting the original program source into instances of skeletons. The role of program shaping is to prepare the source code for the introduction of parallel skeletons: pattern discovery identifies expressions that may have a large variety of syntactic forms, but only a few of these forms are supported by the transformation system. It is therefore necessary to shape the code into an appropriate canonical form before the transformations can be applied. This two-stage approach has the advantage of ensuring that the transformations are simple and understandable, and it also allows the programmer to supervise the process, undoing/redoing changes if required. Moreover, it is much easier to verify preservation of both functionality and behaviour if our transformations and their correctness proofs are built as compositions of smaller and simpler steps.

Program shaping transformations can be classified from two different points of view. Firstly, we have to take into consideration the kind of the algorithmic skeleton that is assigned to the candidate. It defines the syntactic form to which we have to shape the given code fragment. For instance, a transformation that introduces a farm as a *Skel* library call is only applicable on list comprehensions fulfilling the following requirements: the head expression is a function call to a unary function; it has exactly one generator; and it has no filter. In the case of pipe we have different needs and criteria. Secondly, we can distinguish shaping transformations according to the syntactic category of the discovered source code fragment to be shaped. These categories appeared also in Section 4 where we showed how these constructs can be identified. Here we have to handle the various syntactic structures differently in the shaping phase.

5.1 Introducing Parallelism

Wrangler refactorings are used to introduce and tune parallelism using the *Skel* library [9]. In this paper, we will mainly use the *introduce farm* and *introduce pipeline* refactorings, described below. Each refactoring is given as a transformation rule plus any associated preconditions, operating over the source program's abstract syntax tree (AST).

$$Refactoring(x_0, ..., x_n) = \{Rule \times \{Condition\}\}$$

where $x_0, .., x_n$ are the arguments to the refactoring. The refactoring rules are defined as functions over nodes that represent expressions in the AST:

$$\mathcal{E}[\![.]\!] :: Expr \rightarrow Expr$$

Quasi-quotes are used to denote code syntax, so that e.g. $[\![\texttt{f = e}]\!]$ denotes a function in the AST of the form `f = e`.

Pipeline Introduction. The rule for the *Introduce Pipeline Refactoring* is shown below.

$$\begin{array}{l} PipeIntro(\rho, \texttt{e}) = \\ \quad \mathcal{E}[\![[\, \texttt{f}_1\ (\ \texttt{f}_2\ (\ ...\ \texttt{f}_n(\texttt{Input}\ )...\ ))\ ||\ \texttt{Input} \leftarrow \texttt{Inputs}\ ]\,]\!] \Rightarrow \\ \qquad [\![\, \texttt{skel} : \texttt{do}([\{\texttt{pipe}, [\{\texttt{seq}, \texttt{fun}\ ?\texttt{MODULE}:\texttt{f}_n/1\},\ ...\ , \\ \qquad\qquad\qquad\qquad \{\texttt{seq}, \texttt{fun}\ ?\texttt{MODULE}:\texttt{f}_2/1\}, \\ \qquad\qquad\qquad\qquad \{\texttt{seq}, \texttt{fun}\ ?\texttt{MODULE}:\texttt{f}_1/1\}]\}], \texttt{Inputs})\,]\!] \\ \qquad \{\texttt{skel} \in \texttt{imports}(\rho), \texttt{run} \in \rho, \texttt{pipe} \in \rho,\ \texttt{seq} \in \rho, \\ \qquad\ \texttt{f}_1, \texttt{f}_2, , ..., \texttt{f}_n \in \rho, \texttt{Inputs} \in \rho\} \end{array} \quad (1)$$

The rule takes an environment, ρ, and an expression, `e`, denoting the syntax phrase to be matched. It matches an Erlang list comprehension, e.g.:

```
[ f1 (f2 (f3( Input ))) || Input <- Inputs ]
```

and transforms it into a *Skel* pipeline:

```
skel:do([{pipe, [{seq,fun f3/1},
                 {seq,fun f2/1}
                 {seq,fun f1/1}]}], Inputs)
```

Here, the pipeline has three stages, where the first stage executes `f1`, the second, `f2` and the third, `f3`. The input stream to the pipeline is preserved as a list, `Inputs`. The preconditions to the refactoring ensure that the newly introduced skeletons, `pipe` and `seq`, all three stages of the pipeline, and the list of inputs, `Inputs`,

are all in scope. We omit the module macro, `?MODULE`, in this and following examples for clarity.

Farm Introduction. The refactoring rule for the *Introduce Task Farm Refactoring* is shown below:

$$
\begin{array}{l}
\mathit{FarmIntro}(\rho, \mathtt{e}, \mathtt{Nw}) = \\
\quad \mathcal{E}[\![[\, \mathtt{f(\ Input\ )} \mid\mid \mathtt{Input} \leftarrow \mathtt{Inputs}\ ]]\!] \Rightarrow \\
\quad [\![\mathtt{skel : do([\{farm, [\{seq, (fun\ ?MODULE : f/1\}], Nw\}],} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{Inputs})]\!] \\
\qquad \{\mathtt{skel} \in \mathtt{imports}(\rho), \mathtt{run} \in \rho, \mathtt{seq} \in \rho, \mathtt{farm} \in \rho\}
\end{array} \tag{2}
$$

The rule takes three parameters: an environment, ρ, an expression, `e`, and the number of workers for the farm, `Nw`. The refactoring rule matches against a list comprehension, converting it into a task farm. If the farm worker is a function composition, then an anonymous function is introduced to pass the tasks into the `farm`. For example, suppose that we have the following list comprehension:

```
[ f ( g(Input) ) || Input <- Inputs ]
```

The refactoring will transform this into:

```
skel:do([{farm , [{seq , (fun(Input) ->
    f ( g(Input) ) end)}] , NW}] , Inputs)
```

5.2 Program shaping Example: matrix multiplication

As shown above, a *farm* skeleton can only be introduced using *FarmIntro* for a list comprehension that has precisely one generator, and whose head expression refers to a unary function. Unfortunately, this is not the case for the matrix multiplication example of Section 4.1: the list comprehension in `mult_matrix/2` contains two generators, and its head expression is also unsuitable. Program shaping aims to fix this by joining the generators and altering the head expression. After performing the shaping transformations, we obtain the following code:

```
mult_matrix(Rows, Cols) ->
  NV1 = [{R,C} || R <- Rows, C <- Cols],
  [fun({R,C})->
     scalar_product(R,C)
   end(NV2) || NV2 <- NV1 ].

scalar_product(R, C) ->
  lists:sum([fun({A, B}) ->
               mult_scalar(A,B)
             end(NV1) || NV1 <- lists:zip(R,C) ]).

mult_scalar(A, B) -> A*B.
```

Here, `NV1` is bound to a newly introduced list comprehension whose generators are copies of the original ones. The two generators in the original list comprehension are replaced by a single new generator which binds `NV2` to the values drawn from this new comprehension. The binary functions in the head of the list comprehensions have also been replaced by anonymous unary functions.

5.3 Shaping other syntactic forms

Calls to `lists:map/2` or `lists:filter/2` can obviously be rewritten as list comprehensions, that can then be parallelized as shown above. Examples are shown in Figures 4 and 5. The latter example also shows that there may be more ways to perform shaping. Although `my_filter_v1` and `my_filter_v2` are functionally equivalent, we would rather shape expressions referring to `lists:filter/2` into the expression that belongs to the `my_filter_v2` function, in the second way, since this would allow

```
my_map(List) -> lists:map(fun worker/1, List).

my_map_shaped(List) -> [worker(Item) || Item <- List].
```

Figure 4. Shaping a call to `lists:map/2`

```
my_filter(List) -> lists:filter(fun pred/1, List).

my_filter_v1(List) -> [Item || Item<-List, pred(Item)].

my_filter_v2(List) ->
    lists:append([ case pred(Item) of
                       true -> [Item];
                       false -> []
                   end || Item <- List ]).
```

Figure 5. Two ways to shape a call to `lists:filter/2`

us to evaluate the potentially computationally-intensive `pred/1` function in parallel. There are many other syntactic forms that can be transformed into a list comprehension, and each of them requires different shaping transformations. In our transformation system, every shaping transformation is responsible for dealing with exactly one specific syntactic form (e.g., a call to `lists:filter/2`). It follows that the pre-conditions of the shaping transformations are disjoint.

5.4 Transformation rule for shaping map-like recursive functions

To shape a recursive function that has been identified as map-like by the analysis of Section 4, the following idea is used. Given a function definition $f(\overline{V}_1, L, \overline{V}_2) \rightarrow \varepsilon^+$, we generate its kernel function $f'(\overline{V}_1, I, \overline{V}_2)$ by substituting every occurrence of L by the improper list $[I|\mathtt{ok}]$, then substituting every occurrence of $\mathrm{Head}(L)$ by I, and finally, every occurrence of $\mathrm{Tail}(L)$ by the atom `ok`. Furthermore, we replace the return expression ϱ of each recursive execution path of f by $\mathtt{hd}(\varrho)$. (After these substitutions some clean-up transformations might be applied to make the code more readable.) The invocations $f(\overline{p}_1, \ell, \overline{p}_2)$ that are to be shaped, are then transformed into calls to `lists:map/2` using the kernel function f', i.e. $\mathtt{lists:map(fun(I)->}f'(\overline{p}_1, \mathtt{I}, \overline{p}_2)\ \mathtt{end,}\ \ell)$.

5.5 Composite transformations

When the user selects a pattern candidate in the *Pattern Candidate Browser*, the corresponding *transformation sequence* is passed to the **PaRTE** transformation system. A transformation sequence comprises a number of *transformation components*; *RefactorErl* equips each component with all the necessary information that is required to perform the transformation. Each transformation component uniquely identifies the source code fragment that needs to be transformed, as well as the skeleton to be introduced, and any necessary configuration parameters (such as the number of workers in a farm). **PaRTE** applies a transformation sequence as an atomic *composite refactoring* [21] using *Wrangler*. The primary effect of a transformation component is determined by the kind of the algorithmic skeleton to be introduced. However, a transformation component may itself still involve multiple steps, including program shaping and introducing the corresponding *Skel* library call. Therefore, a transformation component might also be implemented as a composite refactoring. Shaping changes the selected source code fragment into a functionally equivalent syntactic structure, on which the *Skel* library call can be introduced directly.

The transformations have been implemented as separate refactorings using *Wrangler*'s refactoring DSL [20]. When we combine

these transformations into a composite one, however, we face an annoying technical problem. It is likely that a transformation sequence needs to transform two or more code fragments located in the same source file. When our analyses compute the position of the code fragments to transform, we only know these positions relative to the original source code. When we apply a transformation, the source code changes, and the positions required by subsequent transformations may become invalid. Interestingly, it is not easy to update these positions after the execution of each transformation. While *Wrangler* allows us to track the position of function definitions between steps of a composite transformation, unfortunately, this is not sufficient in our case since most of our transformations address expressions. Moreover, it is not generally possible to simply locate an expression using its syntactic structure, since there may be many similar expressions, and we do not want to transform all of them. The solution we propose is the following. Before applying the transformation sequence, we iterate over the expressions and wrap them into a call to an imaginary function with a unique name. When we actually perform the transformation sequence, we can use these marker calls to identify the expressions to be transformed, rather than using the possibly invalid position information. Following the transformation sequence, the dummy wrapper calls can be easily removed. The main drawback of this solution is that these markers can be observed by the programmer whilst they are supervising the transformation sequence.

6. Worked Example

We return to the image merge example first given in Section 2. For this example, we will first apply our pattern discovery process to find a suitable pattern candidate for parallelisation. This candidate will then undergo a program shaping phase, restructuring it so that it is amenable for parallelisation via a third stage, where we use parallel refactoring to introduce the parallelisation. The image merge operation takes a list of image pairs, merging each pair. This operation is split into two stages, corresponding to the `readImage/1` and `convertMerge/1` functions.

Stage 1: Pattern Discovery Applying the pattern discovery tool to the image merge example, the call to `lists:map/2` in the top-level definition will be highlighted as a possible pattern candidate:

```
merge(Images) ->
    lists:map(fun(Y)
       -> convertMerge(readImage(Y)) end, Images).
```

Stage 2: Program Shaping Having selected the call to function `lists:map/2`, we can now apply program shaping to refactor it into a list comprehension. This will make our pattern candidate amenable to parallelisation. The result of the program shaping is:

```
merge(Images) ->
    [convertMerge(readImage(Y)) || Y <- Images].
```

Stage 3: Parallel Refactoring Since the list comprehension applies the same operation to each of its inputs, we can now refactor this function to introduce a call to *Skel*. The resulting transformation introduces parallelism without changing the shape of the expected output. Several equivalent configurations can be produced by this process: the final source code depends upon the order in which individual primitive refactorings are applied. We first consider the immediate application of *IntroPipe* to the list comprehension resulting from Stage 2. This produces:

```
merge(Images) ->
    skel:do([{pipe,
              [{seq, fun readImage/1},
               {seq, fun convertMerge/1}]}],
            Images).
```

The refactoring replaces the list comprehension with a two-stage pipeline. Whilst some parallelism is introduced this way, as highlighted by the *Pattern Candidate Browser*, it will give only a limited speedup. We note that a maximum of only two workers can be active at any one time. Since the first stage generally requires less time to complete than the second stage, only one worker will generally be active. To introduce more parallelism we repeatedly apply *IntroFarm* to each stage of the pipeline under the guidance of the *Pattern Candidate Browser*. This results in the following:

```
merge(Images) ->
    skel:do([{pipe,
              [{farm,
                [{seq, fun readImage/1}],
                NW},
               {farm,
                [{seq, fun convertMerge/1}],
                NW}]}],
            Images).
```

We retain the two-stage pipeline seen previously, but now each stage is a task farm of `NW` workers. Hence, both stages now distribute their inputs to their workers, which apply their respective functions in parallel. With this configuration we achieve greater parallelism than the previous configuration, so potentially improving performance gains. An alternative (and better) approach shown by the *Pattern Candidate Browser*, is to instead immediately introduce a task farm at the top-level. Applying *IntroFarm* to the original list comprehension results in the following.

```
merge(Images) ->
    skel:do([{farm,
              [{seq, fun(Y) ->
                          convertMerge(readImage(Y))
                end}],
             NW}],
            Images).
```

The list comprehension is transformed into a task farm, with each worker applying an anonymous function. This function returns the result of the composed functions found in the original list comprehension. We might expand this anonymous function using `intro_pipe`, which results in the following:

```
merge(Images) ->
    skel:do([{farm,
              [{pipe,
                [{seq, fun readImage/1},
                 {seq, fun convertMerge/1}]}],
              NW}],
            Images).
```

Here, each worker in the task farm involves a nested parallel pipeline. This potentially streamlines each worker by ensuring that as each `convertMerge/1` operation finishes, the next image pair has already been loaded into memory by `readImage/1`, and so may be processed immediately.

6.1 Results

As shown in Figure 6, we have measured speedups for four parallel configurations of the image merge operation. We report absolute speedups against the original sequential version. All measurements have been made on a 2.4GHz 24-core, dual AMD Opteron 6176 architecture, running Centos Linux 2.6.18-247.el5 and Erlang R16B02, for 100 pairs of 1024x1024 images, and the mean taken over 3 runs. The first configuration, a two-stage pipeline, offers a speedup of 0.8 (i.e. a *slowdown*). This can be attributed to the limited parallelism that has been introduced using *IntroPipe* — at best, only a single read and merge operation can occur at the same time. The slowdown is likely the result of additional overheads introduced. The second configuration, a pipeline with two

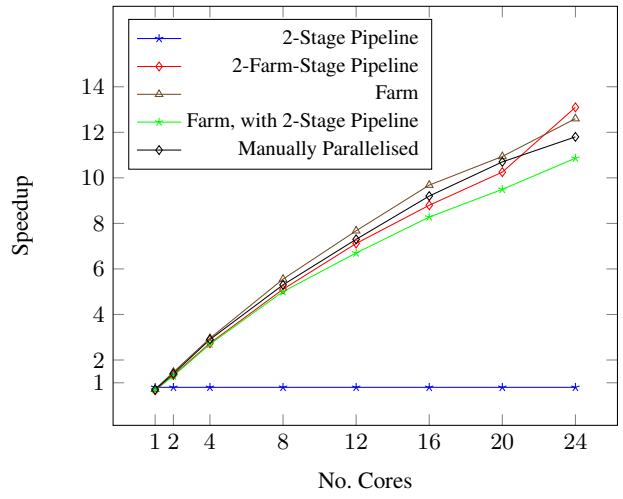


Figure 6. Speedups for Image Merge: 100 pairs of 1024x1024 images.

task farm stages, achieves a maximum speedup of 13.10 on 24 cores, which is the best speedup that we observe. Despite this, we note that, in comparison to the configuration (a single *farm*), this configuration achieves slightly lower speedups on fewer than 24 cores. We suspect that the increase in speedup when all 24 cores are used is either an artefact of the machine itself (perhaps a lucky cache configuration), a result of convenient scheduling by the Erlang Virtual Machine, or possibly a combination of the two. The third configuration, a farm, achieves a maximum speedup of 12.60 on 24 cores, but, as previously mentioned, consistently achieves the best speedups on fewer cores. The fourth and final configuration, a task farm with a nested two-stage pipeline, achieves a maximum speedup of 10.86 on 24 cores. This configuration performs consistently worse than both second and third configurations. Ignoring the simple pipeline, which yields a slowdown, the speedups that we have achieved using automatic pattern discovery and refactoring are comparable to or better than those that we achieved by manual parallelisation, where we achieved a maximum speedup of 11.8 over the original sequential version using 24 cores. The improvements over the manually parallelised version seen here show that whilst naively spawning processes for each input is a simple solution, it is not a good one. Process creation in Erlang is not cheap, hence as the number of tasks increase, the overheads associated with this method increase also. The use of skeletons, in this case Skel, avoids this problem, meaning overheads are kept low, and better speedups achievable. Although we have not done this yet, it may well be possible to extract further performance improvements from the *Skel* configurations by e.g. adjusting the number of workers used by the task farms (we have used 24 workers for all configurations), or by allowing the use of skeletons other than `pipe` and `farm` (e.g. parallel *reduce*). Other preliminary results may be found in [9] and [19].

7. Related work

There has previously been some limited research on developing mechanisms that will analyse the code of applications and discover which parts are amenable to parallelisation, including identifying the parallel patterns that could be applied. Hammacher *et al.* [17] propose a tool for detecting dependencies in Java applications, using dynamic dependency graphs, and suggesting hotspots for parallelisation. Unlike our approach, however, this work doesn't consider patterns as instances of algorithmic skeletons, or provide suggestions for which skeleton to apply. Ferenc et al. [15] describe a code analysis based approach to detect general patterns, and enhance this by Machine Learning based discrimination between closely related patterns and filtering of false candidates. However, this approach only considers instances of Object-Oriented design patterns, and not algorithmic skeletons for parallelisation, as we have done. Molitorisz [28] describes a tool for automatic discovery of parts of the sequential application code that contain master/-worker and pipeline patterns, and the refactoring and performance tuning of these parts to introduce parallelism. The same author proposes the AutoFutures [29] tool that performs static analysis of the source code of Java applications to discover portions of code without data dependencies, and then automatically inserts parallel constructs (futures) that execute them asynchronously in parallel. Unlike the work presented in this paper, however, this work does not take into account general algorithmic skeletons, and moreover does not present possible pattern candidates to the user for further program shaping and parallel refactoring. There has also been some work on using static analysis to discover parallelism bottlenecks, and providing help to the programmer to reshape the program in accordance with such analysis. Most previous work has been done in the context of object-oriented concurrent programs, within the X10 project [24], and in Java [35]. These approaches aim to introduce concurrency rather than parallelism, specifically aiming to provide thread safety and/or re-entrancy. However, the simple special cases that they consider are not integrated into a general approach, and it is unclear how well they work in practice. Moving away from concurrency, some earlier work [30] aims to classify (sequential) programs at the software design level into families of programs, and then to shape programs into the closest such families. This has been extended, for example, to shape programs for pervasive programming applications [33]. However, once again, these approaches do not take a pattern/algorithmic skeleton approach, as we have done, and moreover, they do not present pattern candidate choices to the programmer, for further program shaping or parallel refactoring. Recent work in the field of refactoring for parallelism includes Reentrancer [35], a refactoring tool developed by IBM for making code reentrant, which targets global data by making them thread-safe; work by Dig [14], that introduces concurrency in Java programs, also by targeting thread safety, aiming to increase throughput and scalability; and work by Markstrum *et al.* [25], who use a basic refactoring approach for to introduce a map/reduce-like behaviour in X10. No results are reported by the latter work, however, giving no evidence to the validity or effectiveness of the refactorings. In [9], we have introduced a parallel refactoring methodology for Erlang programs, demonstrating a refactoring tool that introduces and tunes parallelism for Skeletons in Erlang. We have not, however, considered automatic pattern discovery. In [10], we have similarly introduced a small number of basic refactorings to introduce farm and pipeline skeletons into C++ using the FastFlow library. We have also recently extended *HaRe*, the Haskell Refactorer [7], to deal with a limited number of parallel refactorings [8]. This work allows Haskell programmers to introduce data and task parallelism using small structural refactoring steps. However, it does not use pattern-based rewriting or pattern discovery as described here. Finally, SkelML [27] is a skeleton-based parallel compiler for ML that automatically identifies certain forms of parallel skeleton. However, SkelML does not present pattern candidates to the programmer or allow program shaping and parallel refactorings, thus giving the programmer choice and guidance into which parallelisation to perform. The algorithmic skeleton library *Skel* was introduced in [9] with a small group of refactorings to help introduce the library. The library and two of these refactorings have been

used here. The skeletons research community has been working on methods for parallel programming in high-level languages since the nineties [4, 5, 11–13, 16]. This has produced a number of skeletons and skeleton libraries in a range of languages. Other attempts at parallelising Erlang programs can be found in the parallelisation of Dialyzer [1] and a suite of Erlang benchmarks [2], though these use an ad hoc approach.

8. Conclusions

This paper has introduced the *ParaPhrase Refactoring Tool for Erlang* (**PaRTE**), an inter-operable framework that supports the discovery of parallel pattern candidates in Erlang and that provides program shaping refactorings and parallel refactoring. The pattern discovery techniques that we have described allow instances of algorithmic skeletons to be detected *automatically* within Erlang programs, with the results of the detection process presented to the programmer via a *pattern candidate browser* that also reports predicted parallel speedup for each pattern candidate. Refactoring tools are used to *shape* the sequential Erlang programs, in order to make them amenable for the introduction of parallelism. They are also used to transform the program according to the transformation sequences that are defined for each pattern candidate. and algorithmic skeletons. Finally, we have demonstrated the use of **PaRTE** on a simple worked example, an image merge, showing that we can quickly, easily and automatically obtain significant and scalable speedups of around 13 over the original sequential version when running on 24 cores, a better result than our manually parallelised implementation. Our approach can therefore deliver *significant* savings in effort, while giving performance results that are similar to manual approaches. In future, we expect to develop this work in a number of new directions, including investigating the discovery of *near-patterns* in sequential programs, i.e., identifying code fragments that are similar to but exactly consistent with an instance of an algorithmic skeleton. Furthermore, we plan to extend the discovery and refactoring process to a much wider range of skeletons, particularly in the field of computer algebra and physics, in order to demonstrate our approach on a wider range of realistic applications. This will add evidence that our approach is general, useable and scalable.

References

[1] S. Aronis and K. Sagonas. On Using Erlang for Parallelization : Experience from Parallelizing Dialyzer. *Proceedings of the Symposium on Trends in Functional Programming*, 2012.

[2] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A Scalability Benchmark Suite for Erlang/OTP. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, Erlang '12, pages 33–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1575-3. URL `http://doi.acm.org/10.1145/2364489.2364495`.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq Quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. . URL `http://doi.acm.org/10.1145/1159789.1159792`.

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[5] G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proc. of the 5th International Symposium on High Performance Distributed Computing (HPDC'96)*, pages 243–252. IEEE Computer Society Press, 1996.

[6] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, and M. Tóth. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, Tallin, Estonia, October 2011. ISBN 978-9949-23-178-2.

[7] C. Brown. *Tool Support for Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2008.

[8] C. Brown, H. Loidl, and K. Hammond. Paraforming: Forming Haskell Programs using Novel Refactoring Techniques. In *Twelth Symposium on Trends in Functional Programming*, Madrid, Spain, May 2011.

[9] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, and A. Elliott. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming*, pages 1–19, 2013. ISSN 0885-7458. URL `http://dx.doi.org/10.1007/s10766-013-0266-5`.

[10] C. Brown, V. Janjic, K. Hammond, H. Schöner, K. Idrees, and C. W. Glass. Agricultural Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools. In *Proc. PDP '14*. IEEE, 2014.

[11] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-53086-4.

[12] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Comput.*, 30(3):389–406, Mar. 2004. ISSN 0167-8191. URL `http://dx.doi.org/10.1016/j.parco.2003.12.002`.

[13] J. Darlington, Y. Guo, Y. Jing, and H. W. To. Skeletons for Structured Parallel Composition. In *Proc. of the 15th Symposium on Principles and Practice of Parallel Programming*, 1995.

[14] D. Dig. A Refactoring Approach to Parallelism. *IEEE Softw.*, 28:17–22, 2011. ISSN 0740-7459. . URL `http://dx.doi.org/10.1109/MS.2011.1`.

[15] R. Ferenc, Á. Beszédes, L. Fülöp, and J. Lele. Design Pattern Mining Enhanced by Machine Learning. In *Proceedings of the 21st IEEE Int ernational Conference on Software Maintenance (ICSM'05)*, pages 295–304. IEEE, 2005.

[16] M. Hamdan, P. King, and G. Michaelson. A Scheme for Nesting Algorithmic Skeletons. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of the 10th International Workshop on the Implementation of Functional Languages (IFL'98)*, pages 195–211. Department of Computer Science, University College London, 1998.

[17] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling Java Programs for Parallelism. In *Proc. IWMSE '09*, pages 49–55, 2009. ISBN 978-1-4244-3718-4. . URL `http://dx.doi.org/10.1109/IWMSE.2009.5071383`.

[18] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzlez-Vlez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In B. Beckert, F. Damiani, F. S. Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 218–236. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35886-9. URL `http://dx.doi.org/10.1007/978-3-642-35887-6_12`.

[19] V. Janjic, A. Barwell, and K. Hammond. Using Erlang Skeletons to Parallelise Realistic Medium-Scale Parallel Programs. In *Proceedings of the Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems, HLPGPU 2014*, Vienna, Austria, 2014.

[20] H. Li and S. Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, University of Kent, October 2011. URL `http://www.cs.kent.ac.uk/pubs/2011/3171`.

[21] H. Li and S. Thompson. A Domain-specific Language for Scripting Refactorings in Erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 501–515, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28871-5. URL `http://dx.doi.org/10.1007/978-3-642-28872-2_34`.

[22] H. Li, S. Thompson, G. Orosz, and M. Tóth. Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 61–72, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. URL http://doi.acm.org/10.1145/1411273.1411283.

[23] T. Lindahl and K. Sagonas. TypEr: A Type Annotator of Erlang Code. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05, pages 17–25, New York, NY, USA, 2005. ACM. ISBN 1-59593-066-3. . URL http://doi.acm.org/10.1145/1088361.1088366.

[24] S. A. Markstrum and R. M. Fuhrer. Extracting Concurrency via Refactoring in X10. In *Proceedings of the 3rd ACM Workshop on Refactoring Tools*, WRT'09, 2009. URL http://refactoring.info/WRT09/#program.

[25] S. A. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards Concurrency Refactoring for x10. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 303–304, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. . URL http://doi.acm.org/10.1145/1504176.1504226.

[26] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming*. Morgan Kaufmann, 2012. ISBN 978-0-12-415993-8.

[27] G. Michaelson, A. Ireland, and P. King. Towards a Skeleton Based Parallelising Compiler for SML. In *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, 1997.

[28] K. Molitorisz. Pattern-Based Refactoring Process of Sequential Source Code. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 357–360, March 2013. .

[29] K. Molitorisz, J. Schimmel, and F. Otto. Automatic Parallelization Using Autofutures. In *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools*, MSEPT'12, pages 78–81, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31201-4. . URL http://dx.doi.org/10.1007/978-3-642-31202-1_8.

[30] D. Parnas. On the Design and Development of Program Families. *Software Engineering, IEEE Transactions on*, SE-2(1):1–9, March 1976. ISSN 0098-5589. .

[31] M. Pitidis and K. Sagonas. Purity in Erlang. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 137–152, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24275-5. URL http://dl.acm.org/citation.cfm?id=2050135.2050144.

[32] RefactorErl Homepage. http://refactorerl.com, 2014.

[33] S. M. Sadjadi, P. K. McKinley, and B. H. C. Cheng. Transparent Shaping of Existing Software to Support Pervasive and Autonomic Computing. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005. ISSN 0163-5948. . URL http://doi.acm.org/10.1145/1082983.1083086.

[34] M. Tóth and I. Bozó. Static analysis of complex software systems implemented in Erlang. In *Proceedings of the 4th Summer School conference on Central European Functional Programming School*, CEFP'11, pages 440–498, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-32095-8. URL http://dx.doi.org/10.1007/978-3-642-32096-5_9.

[35] J. Wloka, M. Sridharan, and F. Tip. Refactoring for Reentrancy. In *ESEC/FSE '09*, pages 173–182, Amsterdam, 2009. ACM. ISBN 978-1-60558-001-2. . URL http://doi.acm.org/10.1145/1595696.1595723.

Computing and Informatics, Vol. 32, 2013, 1001–??, V 2016-Jun-27

STATIC ANALYSIS FOR DIVIDE-AND-CONQUER PATTERN DISCOVERY

Tamás Kozsik

Eötvös Loránd University, Budapest, Hungary
e-mail: kto@elte.hu

Melinda Tóth, István Bozó, Zoltán Horváth

ELTE-Soft Nonprofit Ltd., Budapest, Hungary
e-mail: {tothmelinda,bozoistvan,hz}@elte.hu

Abstract. Routines implementing divide-and-conquer algorithms are good candidates for parallelization. Their identifying property is that such a routine divides its input into "smaller" chunks, calls itself recursively on these smaller chunks, and combines the outputs into one. We set up conditions which characterize a wide range of *d&c* routine definitions. These conditions can be verified by static program analysis. This way *d&c* routines can be found automatically in existing program texts, and their parallelization based on semi-automatic refactoring can be facilitated. We work out the details in the context of the Erlang programming language.

Keywords: Erlang, divide-and-conquer, pattern, parallelization

Mathematics Subject Classification 2010: 68-N19

1 INTRODUCTION

Divide-and-conquer is a principle that is at the heart of many useful algorithms in different domains, including searching, sorting, FFT and number theory. By their very nature, these algorithms can be implemented in a parallel way, and be efficiently

executed on a parallel machine. Divide-and-conquer is therefore often perceived as a high-level parallel programming pattern [5].

This fact is recognized in the EU FP7 ParaPhrase project (Parallel Patterns for Adaptive Heterogeneous Multicore Systems), which "aims to produce a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically remapped to meet application needs and hardware availability" [27]. In this project, software tools to facilitate parallel programming are designed, which allow programmers to identify parallelizable components, and to refactor them into occurrences of parallel patterns. These patterns can often be expressed in terms of algorithmic skeletons [5]. The implementation of such skeletons may support the dynamic mapping (and re-mapping) of algorithm components onto available hardware resources, and thus support efficient and adaptive resource usage on a heterogeneous many-core machine [8]. Software developers, even when they are not concurrency experts, can make a very good use of these skeleton implementations, if they are collected in a reusable program library, such as the `skel` library [24] for the Erlang programming language [11].

In this paper we focus on the identification of parallelizable components, viz. *pattern discovery*, in particular the identification of divide-and-conquer structures (or *d&c* for short). Pattern discovery is a technique to use static program analyses to find those fragments of a possibly large code body which exhibit the same behaviour as a pattern; such code fragments are called *pattern candidates*. The pattern discovery technique can be exploited in a software development tool, which may help software developers make good programming decisions. If the tool finds a pattern candidate, it may notify the programmer, who may decide to refactor the code to make the pattern explicit, possibly implementing the pattern with some predefined algorithmic skeletons. The tool may even help to perform the refactoring by applying automated program transformations on the source code under the supervision of the software developer. The refactoring technology to introduce `skel` skeletons is described e.g. in [4, 1].

The ParaPhrase Refactoring Tool for Erlang (a.k.a. PaRTE), offers exactly the above services [2]: software developers can find pattern candidates in existing Erlang code, and refactor them into applications of `skel` skeletons. Moreover, PaRTE is able to estimate the speedup that can be achieved by transforming a candidate to a skeleton-based parallel implementation, and hence prioritize candidates, in order to propose the most promising ones to its user.

The main contribution of this paper is *the technique of d&c candidate discovery, implemented in a static analysis framework for Erlang* (as part of PaRTE). Additionally, the paper *characterizes typical occurrences of the pattern* using small examples of well-known algorithms, and *proves the applicability of the technique* on real-world source code bodies. The presentation of *d&c* pattern discovery is concretized here for a nice and relatively simple language, Erlang, but the technique can be generalized to other languages, or even paradigms.

Although the identification of a *d&c* algorithm in some source code is an interesting problem by itself, its main benefit lies in its capability to facilitate the parallelization of the code. Once we know where to introduce parallelism, we can refactor the code either using a tool, or manually, to turn it into a parallel *d&c*. In this scenario, some additional side conditions must be verified, which ensure that the parallel components of the computation do not interfere with each other. Our ultimate focus on parallelization has an effect also on how we define *d&c*: "degenerated" cases of *d&c* algorithms, i.e. those that should not be turned into a parallel *d&c* implementation, will be exluded in our approach.

At the time of writing, support for the *d&c* pattern is limited in PaRTE. The tool is already able to identify *d&c* pattern candidates using the ideas presented in this paper. However, the introduction of the skeleton-based implementation of the pattern is to be carried out manually, because the completely automated transformations are not available yet.

The rest of the paper is structured as follows. Section 2 elaborates on the concept of pattern discovery. Section 3 points out the characteristics of *d&c* algorithms using a sequence of examples. Section 4 reveals the (Erlang-specific) static analyses required to discover *d&c* candidates. Section 5 describes the evaluation of the presented method. Section 6 discusses related work, and Section 7 concludes the paper.

2 PATTERN DISCOVERY

Pattern discovery is the act of finding pattern candidates in some source code by applying static program analyses. Albeit its applicability is broader, we focus on discovering *parallel* pattern candidates in general, and, in this paper, the *d&c* pattern in particular. Pattern candidates are typically characterized with a *syntactic description* and a set of properties that can be verified with some *semantic analyses*. Therefore, discovery starts with locating certain program structures (e.g. list comprehensions, function compositions, `receive`-expressions in Erlang), and then it filters out irrelevant ones using a combination of different standard analyses (such as control-flow, data-flow and data-dependence analyses), and specific ones (such as side-effect analysis). Finally, a ranking of found candidates can be computed, if a measure of "candidate goodness" is available. For example, in PaRTE a special technique to estimate parallel speedup is used to rank found parallel pattern candidates. In this paper, however, we will not consider this final step, we will focus on the former two.

PaRTE is based upon RefactorErl [3], a static program analysis and transformation framework for Erlang. RefactorErl implements the standard control-flow and data-flow analyses in an Erlang-specific way, taking into account the specificities of a concurrent, dynamically typed, impure functional language (see Section 4.1). RefactorErl also provides a rich representation of Erlang programs on which transformations, e.g. refactorings, are easy to implement.

In a language like Erlang, the main sources of uncertainty in static analyses are higher-order functions and the message-passing style inter-process communication. In the presence of such constructs, higher-order analyses [23] may be necessary to increase the precision of analysis results. However, for our purposes, first-order analyses will be sufficient in most of the cases. Firstly, the code fragments that should be parallelized are likely to contain only computations, and no inter-process communication (such as Erlang/OTP behaviours[1], callbacks etc.). Secondly, higher-order functions and lambdas are less frequently written, or even used, in Erlang than in other functional languages; for instance, list comprehensions appear in code 5 to 10 times more often than the higher-order list processing functions of the Erlang/OTP `lists` module [20].

It is important to understand the ultimate goal of pattern discovery. It will not be built into a compiler in order to form the basis of a completely automatic program transformation (compiler optimization). In contrast, pattern discovery will be built into an integrated development environment (IDE): a software development tool to help its users make good programming decisions. The user of the tool, a software developer, should make those decisions; pattern discovery merely collects information that can be used – or discarded – by the software developer. False positives and false negatives are therefore tolerable. It is clear that in order to be useful, pattern discovery should be as precise as possible, but in an interactive tool there is always a trade-off between preciseness and response time.

The interactive nature of pattern discovery opens up further possibilities. Software developers may pass options to the discovery tool which can be used to filter the search results, and they may re-run the analyses multiple times, with different options, until satisfactory results are obtained. For instance, if the software developer believes that the discovery returns too many false positives for a given Erlang module, he or she can increase the order of the analyses for that particular module, and allow the tool a bit more time, – a few minutes, or even hours – to work.

2.1 Refactoring a pattern candidate

One possible option which can be passed to pattern discovery would exclude candidates that cannot be automatically refactored by the tool. Assume that a software developer only wants to find those candidates that can be trivially transformed into an application of a parallel skeleton library.

For example, consider the following expression, which may appear in the code of a parser for a programming language. It reads, tokenizes and parses a list of source code modules – which can be parallelized in a straightforward manner.

```
1 [ parse(scan(read(Module))) || Module <- Modules ]
```

[1] OTP – a bit misleadingly – stands for Open Telecom Platform, which is "simultaneously a framework, a set of libraries, and a methodology for structuring applications" [11] for Erlang.

This candidate, a list comprehension with a function composition in the head, can be recognized as a pipeline pattern, and easily parallelized using a parallel skeleton library. A pipeline applies n transformations on each element of an input list, where n is the number of processes assigned to this task. Each process takes an item from the previous process in the pipeline, applies its transformation ("stage"), and passes the resulting item on to the next process in the pipeline. This is a parallel programming pattern, since each of the processes can work on different items at the same time. The list comprehension can be transformed into an application of the `skel` skeleton library in the following way.

```
1 STAGE1   = { seq,  fun read/1                    },
2 STAGE2   = { seq,  fun scan/1                    },
3 STAGE3   = { seq,  fun parse/1                   },
4 PIPELINE = { pipe, [STAGE1, STAGE2, STAGE3] },
5 skel:do( [ PIPELINE ], Modules )
```

Skeletons in `skel` are expressed as tuples tagged with specific atoms, such as `seq` and `pipe`. The stages of the pipeline, the three functions, are "sequential components" denoted with `seq` – such sequential components are the basic building blocks of any skeletal descriptions. The skeleton is executed by calling the `skel:do/2` function on the skeleton description and the input list, here `Modules`.

In order to further increase parallelism, one could wrap the pipeline in a task farm, and process e.g. 5 instances of the pipeline in parallel.

```
1 skel:do( [ { farm, [ PIPELINE ], 5 } ], Modules )
```

Note that these transformations are really easy to carry out by a tool. However, many pattern candidates are more hidden, or tangled with other code fragments. Certain candidates can be automatically transformed at once, other candidates need to be reshaped first with a sequence of small refactorings. We presented a methodology of this programmer guided, semi-automatic transformation technique in [1]. In the most complicated cases, manual refactoring of the code might be necessary.

2.2 Examples of parallel patterns

The previous section gave examples of two well-known parallel patterns: the pipeline and the task farm (written in `skel` as `pipe` and `farm`, respectively). Parallel pattern libraries in general, and `skel` in particular, offer some further patterns as well.

Pipeline is used when a function composition is applied on a (possibly long) sequence of data. The functions that make up the composition are deployed in different processes, and hence form the stages of the pipeline.

```
{ pipe, [ Stage | Stages ] }
```

Task farm is suitable to perform an operation on a large data set using a number of processes. Each process takes an element of the input data set, applies the

operation on it, puts the result in the output data set, and recurses until the input is empty. If the input data set is a sequence, care must be taken to produce the output in the same order. A task farm can also operate on a stream of inputs.

```
{ farm, Task, NumOfWorkers }
```

Parallel map is also applied on a large data set. A process decomposes the data set into smaller chunks, further processes perform the necessary operation on these chunks, and the last process collects results and recomposes the output data set.

```
{ map, Task, Decomp, Recomp }
```

Parallel *d&c* is the parallelized execution of a divide-and-conquer algorithm. A divide-and-conquer algorithm is to be applied on a (large) problem, which can be decomposed (divided) into smaller ones. This decomposition can be recursive, until a small-enough subproblem (base case) is reached. An operation (base function) is applied on the base cases, and the results are recombined (following the same recursive structure as for decomposition).

It is important to note that a subproblem can be processed in parallel with the processing of other (e.g. sibling) subproblems, where processing means either applying the base function, or recursively calling the divide-and-conquer operation. Hence we have a parallel pattern of the following structure.

```
{ dc, IsBase, BaseFun, Divide, Combine }
```

Note that the *d&c* pattern can be expressed as a recursive parallel map, as illustrated by the following pseudo-code (based on [19]).

```
1  dc ( IsBase , BaseFun , Divide , Combine ) ->
2   fun ( Data ) ->
3    case IsBase ( Data ) of
4     true  -> BaseFun ( Data ) ;
5     false -> (parmap( dc (IsBase,BaseFun,Divide,Combine),
6                       Divide,
7                       Combine
8                     )
9               )(Data)
10   end
11  end.
```

Practical implementations of the *d&c* parallel pattern may provide ways to bound parallelism. For example, an upper bound of created processes could be given,

```
{ dc, IsBase, BaseFun, Divide, Combine, MaxProcesses }
```

or subproblems satisfying a given property (e.g. subproblems of a given size) could be processed sequentially (i.e. with a sequential divide-and-conquer operation) rather than recursively with the parallel divide-and-conquer operation.

```
SEQ_DC = { seq_dc, IsBase, BaseFun, Divide, Combine },
PAR_DC = { dc, IsSeq, SEQ_DC, Divide, Combine }
```

From all these patterns, we focus on the parallel *d&c* pattern in this paper. Moreover, we are interested here mainly on *d&c* pattern *discovery*.

3 CHARACTERIZATION OF *D&C*

Divide-and-conquer algorithms recursively divide a problem into subproblems, solve those subproblems independently, and combine the results. Mou and Hudak gave an algebraic model in [15], which characterizes a large class of *d&c* algorithms as pseudomorphisms: a "divacon" is a function f that is defined as the function composition

$$c \circ h \circ (\text{map } f) \circ g \circ d$$

on "non-basic inputs" (i.e. on inputs that are intended to be further divided), where d is a divide function, c is a combine function, and g and h are "adjust functions". The last constituent of a divacon is the "base function", which is applied on basic inputs. Without going into details on further requirements on divacons, we emphasize that f is applied multiple times on independent inputs within its own definition, as implied by "map f". Note that this simple characterization considers functions like `map` or `fold` as divacons – from our perspective these examples are, in fact, degenerate cases, since we prefer to recognize them as other patterns.

Let us remember the essence of the above structural description, but now we should change the viewpoint. Rather than considering the algebraic characterization, which is a canonical form of the pattern, one could examine a range of interesting syntactic occurrences of the pattern, which can appear quite naturally, or maybe rather unnaturally, in a functional program. We will go through some code examples, and see what we can learn from them – in order to develop a pattern discovery technique that works well enough in practice.

Quicksort. The most obvious, syntactically recognizable, form of a *d&c* algorithm is when we find multiple recursive calls in the body of a function.

```
1 qs(List)->
2   case List of
3     [] -> [];
4     [H | T] ->
5         {SubList1, SubList2} =
6                 lists:partition(fun(X)-> X < H end, T),
7         qs(SubList1) ++ [H] ++ qs(SubList2)
8   end.
```

The functional-style quicksort algorithm divides a list to be sorted into two sublists based on a pivot element (line 6), recursively sorts both lists, and concatenates the results (line 7).

Mergesort. In its most straightforward form, mergesort is written very similarly to quicksort – the main difference between the two is that in quicksort the divide-phase is the computationally more expensive phase, while in mergesort it is the combine-phase. Another, maybe less natural phrasing of mergesort shows that a case with two recursive calls is just a special case of the `map`-form.

```
1  ms(L) ->
2    case L of
3      [] -> [];
4      [H] -> [H];
5      [H | T] ->
6        {SubList1, SubList2}=
7          lists:split((length(L) div 2), L),
8        [L1, L2] =
9          lists:map(fun ms/1, [SubList1, Sublist2]),
10       merge(L1, L2)
11   end.
12
13 merge([], L2)-> L2;
14 merge(L1, []) -> L1;
15 merge([H1|T1], [H2|T2] = L2) when H1 < H2->
16     [H1 | merge(T1, L2)];
17 merge([H1|T1] = L1, [H2|T2]) ->
18     [H2 | merge(L1, T2)].
```

What happens, if we extract the recursive calls to `ms/1` and the call to `merge/2` into a separate function?

```
1  ms(L) ->
2    case L of
3      [] -> [];
4      [H] -> [H];
5      [H | T] ->
6        {SubList1, SubList2}=
7          lists:split((length(L) div 2), L),
8        sort_and_merge(SubList1, SubList2)
9    end.
10
11 sort_and_merge(L1,L2) ->
12   merge( ms(L1), ms(L2) ).
```

When pattern discovery tries to identify recursive calls, it must take into account that recursion may happen indirectly, through other functions, as well.

Karatsuba. Our next *d&c* example is the well-known fast multiplication method for large integers. The integer numbers are represented as bit-strings. When a problem is divided into subproblems, smaller bit-strings are constructed (half the size of the original bit-strings). This example demonstrates a case when the original problem is divided into more than two subproblems: the `karatsuba/2` function calls itself three times in its body (lines 12–14). Moreover, the problem is represented with two, and not only one formal argument.

For brevity, we skip the definition of the `add/2`, `sub/2` and `shift/2` functions, which add, subtract and left-shift integers represented as bit-strings.

```
karatsuba(Num1, Num2) ->
  S1 = bit_size(Num1),
  S2 = bit_size(Num2),
  case {Num1, Num2} of
    {<<0:1>>, _} -> <<0:S2>>;
    {_, <<0:1>>} -> <<0:S1>>;
    {<<1:1>>, _} -> Num2;
    {_, <<1:1>>} -> Num1;
    _            ->
      M = max(S1, S2),
      M2 = M - (M div 2),
      <<Low1:M2/bitstring, High1/bitstring>> = Num1,
      <<Low2:M2/bitstring, High2/bitstring>> = Num2,
      Z0 = karatsuba(Low1,Low2),
      Z1 = karatsuba(add(Low1,High1),add(Low2,High2)),
      Z2 = karatsuba(High1,High2),
      add(  add(shift(Z2, M2*2), Z0),
            shift(sub(Z1,add(Z2, Z0)), M2)  )
  end.
```

Radix sort. The common property of the before mentioned algorithms is that all of them contain multiple, but a small number of, recursive calls. However, a function can call itself many times, and this may not be conveniently expressed with a sequence of recursive calls in the code of the function – the canonical `map`-form discussed above seems more appropriate. The following `sort/2` function definition nicely exhibits this canonical form. Note that `sort/2` does not call itself directly, but rather through the higher-order `lists:map/2` function. This kind of implicit recursion should be also handled properly by the pattern discovery analyses. One way to achieve this is to build special knowledge about the functions of the `lists` module into the analyses.

```
sort(  [],    _  ) ->  [];
sort( [V],    _  ) -> [V];
sort(List, Level) ->
  Buckets = divide(List,Level),
  SortedLists = lists:map( fun(B) -> sort(B,Level+1) end,
                           Buckets ),
  lists:append(SortedLists).
```

Erlang programmers prefer list comprehensions to `lists:map/2`. The third clause of `sort/2` can be written considerably shorter with this language construct. A recursive call in the head of a list comprehension is the sign of a *d&c* function with very high probability.

```
sort(List, Level) ->
  lists:append([sort(B,Level+1) || B<-divide(List,Level)]).
```

Minimax. The depth-limited minimax algorithm is our next example. The presented implementation uses two mutually recursive functions, where both functions call the other one multiple times. Moreover, it can be the case that the number of children is not fixed for all nodes, and hence the functions call each other in varying number of times at different levels of the algorithm. Note that both of the mutually recursive functions can be regarded as *d&c* functions.

```
mm_max(Node, Depth) ->
  case Depth == 0 orelse terminal(Node) of
    true  ->
       value(Node);
    false ->
       lists:max([mm_min(C,Depth-1)||C <- children(Node)])
  end.
mm_min(Node, Depth) ->
  case Depth == 0 orelse terminal(Node) of
    true  ->
       value(Node);
    false ->
       lists:min([mm_max(C,Depth-1)||C <- children(Node)])
  end.
```

3.1 Non-trivial occurrences of the pattern

So far we have seen some natural program code structures, which implement *d&c* algorithms. Our real-life examples revealed some classes of *d&c* pattern candidates. We have learnt that a function may call itself explicitly multiple times within its own body, within a called function, or as part of a mutually recursive set of functions. The multiple recursive calls may appear lexically in the code, but they may take place due to an iterative structure, such as a list comprehension, or to a special higher-order function, such as `lists:map/2` as well.

However, pattern discovery should be able to cope with trickier examples, too. In some real-world code the programmer may write a function, which has the same recursion structure as `lists:map/2`, but maybe tangled with some other functionality. In general, it is possible to rewrite these functions with `map`, and hence make them more elegant, but pattern discovery should be able to find *d&c* functions in inelegant code as well.

```
sort(List, Level) ->
    lists:append(conquer(divide(List,Level),Level)).

conquer([],Level) ->
    [];
conquer([B|Bs], Level) ->
    [sort(B,Level+1) | conquer(Bs, Level)].
```

In [1] we have defined an analysis to discover "map-like functions" – those that basically work as `lists:map/2`, just like the above `conquer/2` function. As we have

already noted, discovery of *d&c* candidates must investigate `lists:map/2` calls, but it must be able to recognize map-like functions as well.

The next complication arises when the `mapping` of the recursive calls and the combine phase get tangled into a "fold-like function", like, for instance, in the next code fragment. If pattern discovery can identify fold-like functions, this knowledge can be exploited in *d&c* discovery as well.

```
sort(List, Level) ->
    conquer(divide(List,Level),Level).

conquer([],Level) ->
    [];
conquer([B|Bs], Level) ->
    sort(B,Level+1) ++ conquer(Bs, Level).
```

One can construct even more contrived *d&c* functions, based on the ideas from the following example.

```
x(P) ->
  ...
  r(fun x/1, partition(P))
  ...

r(F,Q) ->
  A = some_part_of(Q),
  B = some_other_part_of(Q),
  ...
  C = F(A),     % A does not depend on D
  D = r(F,B),   % B does not depend on C
  ...
```

Function `x/1` calls `r/2` (line 3); `r/2` is recursive (line 11), and contains a call to `x/1` (line 10). Now both `x/1` and `r/2` are *d&c* candidates (if some data independence side-conditions hold). Here `r/2` is responsible for an iterative call of `x/1`, and can be regarded as a generalization of map-like functions.

3.2 Non-trivial occurrences of the non-pattern

Strictly speaking, every recursive function can be considered as *d&c*. The structure of the simplest recursive definitions is the following.

```
f(Problem) ->
  case base_case(Problem) of
    true  -> basic_function(Problem);
    false -> SubProblem = divide(Problem),
             SubSolved = f(SubProblem),
             combine(SubSolved)
  end.
```

This corresponds to the characterization of *d&c* definitions, especially if line 5 is replaced with the equivalent

```
5               [SubSolved] = lists:map(fun f/1, [SubProblem]),
```

line. However, we should regard similar definitions as degenerated cases of *d&c*. In contrast, we want *d&c* discovery to focus only on the really profitable candidates, ones that can benefit from pattern-based parallelization. Moreover, we expect *d&c* discovery to choose the best pattern for a candidate. If a function is map-like, pattern discovery should propose a task farm or a map pattern, and if it is fold-like, a reduce pattern is completely suitable – although both map-like and fold-like functions can be considered as special cases of *d&c* (indeed, [15] uses imbalanced reduce as an example of divacon functions).

```
1  maplike(List) ->
2    case isempty(List) of
3     true  -> [];
4     false ->
5        [SubList] = [tail(List))],
6        [SubSolution] = lists:map(fun maplike/1, [SubList]),
7        [someunaryoperation(head(List)) | head([SubSolution])
8    end.
9
10 foldlike(List) ->
11   case isempty(List) of
12    true  -> defaultvalue;
13    false ->
14      [SubList] = [tail(List))],
15      [SubSolution] = lists:map( fun foldlike/1, [SubList]),
16      somebinaryoperation(head(List), head([SubSolution]))
17   end.
```

Intuitively, pattern discovery should identify a function as *d&c* candidate, if it calls itself more than once during the execution of a single instance of its body – but we shall make this precise in Section 4.

Another important aspect in the analysis of *d&c* candidates is how we deal with execution paths. If a function calls itself on different execution paths, as in the following definition of `binsearch/4`, we should not consider it as *d&c*.

```
1  binsearch(Array,Pattern) ->
2      binsearch(Array,0,array:size(Array)-1,Pattern).
3  binsearch(Array,Lower,Upper,Pattern) when Lower =< Upper ->
4      H = (Lower+Upper) div 2,
5      Val = array:get(H,Array),
6      if
7        Val < Pattern -> binsearch(Array,H+1,Upper,Pattern);
8        Val > Pattern -> binsearch(Array,Lower,H-1,Pattern);
9        true          -> true
10     end;
11 binsearch(_,_,_,_) -> false.
```

Moreover, depending on the capabilities of the control flow analysis, *d&c* discovery may be able to tell the difference between the following two definitions: a strangely written `binsearch/4` function (which is non-*d&c*) and the quicksort on non-empty lists (which is).

```
binsearch(Array,Lower,Upper,Pattern) when Lower =< Upper ->
  H = (Lower+Upper) div 2,
  Val = array:get(H,Array),
  (Val == Pattern)
   orelse
  (Val<Pattern andalso binsearch(Array,H+1,Upper,Pattern))
   orelse
  (Val>Pattern andalso binsearch(Array,Lower,H-1,Pattern));
binsearch(_,_,_,_) -> false.

qs( [H|T] ) ->
    {List1, List2} = lists:partition(fun(X)-> X<H end, T),
    Left  = if length(List1) > 1 -> qs(List1);
               true              -> List1
            end,
    Right = if length(List2) > 1 -> qs(List2);
               true              -> List2
            end,
    Left ++ [H] ++ Right
  end.
```

In our final example, we reimplement the `qs/1` function again. Note that this new implementation can be slightly faster, because `qs/1` is now defined in terms of a tail-recursive helper function, `qs/2`.

```
qs(List) -> lists:reverse(qs([],[List])).

qs(Result, [])              ->
        Result;
qs(Result, [[] | Lists])    ->
        qs(Result,Lists);
qs(Result, [[H] | Lists])   ->
        qs([H|Result],Lists);
qs(Result, [[H|T] | Lists]) ->
        {SubList1, SubList2} =
             lists:partition(fun(X)-> X < H end, T),
        qs(Result,[SubList1, [H], SubList2 | Lists]).
```

Theoretically, this is the same computation: the two recursive calls are simply replaced with an accumulator in the second argument of `qs/2`, which emulates the call stack. However, we shall not consider this definition *d&c* anymore. The recursion structure is completely changed: the `qs/2` function does not call itself "iteratively", as `qs/1` did in the original implementation. This kind of semantic equivalence is out of the scope and the capabilities of our *d&c* candidate discovery.

4 CANDIDATE DISCOVERY FOR *D&C*

In order to find *d&c* candidates, function definitions in the program source code shall be analysed. This section describes the required analyses: some of them are standard, general analyses, such as the construction of a control-flow graph, others are specific to *d&c* discovery. The description assumes that an abstract syntax tree (AST) is already available for the program text to be analysed. In the context of the Erlang language, this AST contains "forms" (such as `module` declarations, `import` and `export` clauses, function definitions etc.), function and expression clauses (e.g. `case` and `receive` clauses), and expressions.

Section 4.1 summarizes how standard analyses map to a functional language like Erlang. Then, Section 4.2 presents rules for the identification of *d&c* candidates. These rules rely on the results of the standard analyses. Since these rules may be computationally expensive, in practice an approximation of these rules may be very useful. This idea is elaborated in Section 4.3.

4.1 Standard static analyses customized for Erlang

Now we present a brief overview of some widely applied analyses, and show how these can be interpreted for Erlang programs. More details on these analyses, including a formal discussion, can be found in e.g. [28].

The "zeroth-order" analyses can be performed on an AST. It is well-known that the results of one analysis can be used to refine the input for another, which in return will provide more precise analysis results. Therefore, one can execute an analysis sequence repeatidly – theoretically, until a fixed point is reached. This way higher order analysis results are obtained. Since the iterative execution of all analyses is definitely expensive for large programs, in practice first-order analyses are already considered good enough. However, we are not restricted to lower order analyses. As explained in Section 2, we envision an interactive software analysis tool, in which the order of the analysis can be a customizable parameter, and the user can analyse "seemingly interesting" parts of the source code more deeply and precisely.

4.1.1 Control-flow analysis

In a functional language, the execution of a program is the evaluation of its expressions. The control-flow analysis should therefore discover the evaluation order of expressions. In a non-lazy language, like Erlang, this evaluation order can be computed more easily that in a lazy one.

The control-flow analysis starts from an entry point of a program. In Erlang, this is a call of an arbitrary exported function: a function that is visible from outside of its containing module.

Based on the above observations, the *inter-procedural control-flow graph* (CFG) of an Erlang program from an exported function f is denoted as $G_{CF}(f)$. This is a directed graph with labeled edges. Its nodes are the nodes in the AST of

the program, plus some auxiliary nodes. Moreover, its edges correspond to the evaluation order of expressions. Edge labels are conditions, as seen below.

For each function g which is called during the evaluation of the main entry point (i.e. f), auxiliary graph nodes are created, with the following intended meaning.

- $start_g$ represents the start of the evaluation of g,
- end_g represents the end of the evaluation of g,
- $call_g^c$ represents the calling of g at a call site c,
- ret_g^c represents the returning from g at a call site c.

$G_{CF}(f)$ is constructed as a union of the intra-procedural control-flow graphs of the called functions, connected through the above auxiliary graph nodes. Roughly, an intra-procedural control-flow graph is constructed by visiting the AST of its body expressions in post-order, respecting the natural "subexpressions first" evaluation order. At branching expressions the control-flow graph contains branches as well, and the appropriate conditions are added as edge labels. (All other edge labels are considered *true*.) The intra-procedural control-flow graph of a function g contains the auxiliary nodes $start_g$ and end_g: there is an edge from $start_g$ to the first pattern of the first function clause, and there are edges from each return expression of g to end_g. (If we regard the intra-procedural CFG as a partial order over the constituting expressions, $start_g$ is the bottom, and end_g is the top value of the partial order.)

If a call to a function g occurs in any of the intra-procedural CFGs (including that of g), the graph node c representing the call is replaced with two new nodes: $call_g^c$ and ret_g^c. All incoming edges (a, c) (for some other node a) are replaced with edges $(a, call_g^c)$, and all outgoing edges (c, a) are replaced with edges (ret_g^c, a). Moreover, the edges $(call_g^c, start_g)$ and (end_g, ret_g^c) are also present in $G_{CF}(f)$.

There is one more important concept related to the control-flow graph, namely execution paths (EP). An execution path is a path in $G_{CF}(f)$, which may visit an edge multiple times. A finite execution path terminates with the end_f node, but infinite execution paths are also possible. Given a $G_{CF}(f)$, the set of execution paths starting from a node v will be denoted by $EP(v)$.

4.1.2 Data-flow analysis

Similarly to the inter-procedural control-flow graph, the *data-flow graph* G_{DF} of an Erlang program can be defined as a directed graph with labeled edges. Again, the nodes are the nodes of the AST – representing the (sub)expressions of the analyzed program.

The graph edges describe the flow of data. An edge from u to v represents the fact that there may be an execution of the program where the value of the expression u flows into expression v. Labels on the edges characterize the different types of data flow.

- If the value of u provides the value for v, the *flow* label is used. For example, the

actual parameter of a function *flow*s into the formal parameter, or the right-hand side of a match expression *flow*s into the left-hand side expression.

```
{A,B} = {X+Y,Y}
```

In this expression `{X+Y,Y}` *flow*s into `{A,B}`.

- If v is a tuple, list etc. expression, and u provides the value of a substructure (i.e. an element of a tuple/list, the tail of a list etc.), the label is *construct* – more specifically, it is $construct_i$, $construct_{tail}$ etc. For the match expression above, for example, there exists a $construct_1$ edge from `X+Y` to `{X+Y,Y}`.
- If u is a tuple, list etc. expression, and v extracts a component, then the label *select* is used, e.g. $select_i$, $select_{tail}$ etc. For the match expression above, for example, there exists a $select_1$ edge from `{A,B}` to `A`.
- Other data flow dependencies between expressions are labeled with *dep*, like the edge from `X` to `X+Y` in the example above.

From the *flow*, *construct* and *select* edges of G_{DF}, the *data-flow reaching* relation can be computed. This data-flow reaching computation pairs *construct* and *select* edges. For example, in the match expression above the expression `A` is reachable from `X+Y`, because of the $construct_1$ – *flow* – $select_1$ path in the G_{DF}.

4.1.3 Derived information

Function call graph. Being a functional language, Erlang supports higher-order functions. Therefore, the *function call* analysis depends on the result of the data-flow analysis (and vice-versa). As mentioned above, an approximation can be achieved by applying a fixed (but customizable) number of iterations. The result of the function call analysis is the function call graph, which will be denoted as G_{FC}.

Dependence graph. The control dependence graph (CDG) can be computed from the control flow graph as described in e.g. [16, 17]. Then the CDG is extended with the *dep*-edges of the G_{DF} and the data-flow reaching relation to form $G_D(f)$, the *dependence graph* of the program with entry point f. When a *dep*-edge or a data-flow reaching edge is to be added to the $G_D(f)$ from, or to, a function call expression, then outgoing edges will start from the respective *ret* node of the CFG, and incoming edges will arrive at the *call* node.

A path from u to v in $G_D(f)$ will be denoted by $u \overset{\mathbf{dep}}{\rightsquigarrow} v$.

4.2 Identifying rules for *d&c*

The divide and conquer pattern describes a computation where a problem is recursively *divided* into sub-problems (until a given condition), and after solving the sub-problems the sub-solutions are *combined* to produce the final solution.

As we have seen in Section 3, there are many syntactic forms which express a recursive *d&c*-like function. From semantical point of view, a canonical form can be given as follows.

```
1 cdc(Problem) ->
2   case isbase(Problem) of
3     true -> solve(Problem);
4     false ->
5       SubProblems = divide(Problem),
6       SubSolutions  = lists:map(fun cdc/1, SubProblems),
7       combine(SubSolutions)
8   end.
```

Therefore, we are looking for a function, which

- has at least one parameter, defining the problem to solve; and
- has, or triggers, multiple recursive calls in its body;

furthermore, these recursive calls satisfy the following properties:

- their actual parameters do not depend on the result of (other) recursive calls;
- their actual parameters depend on the formal parameters of the function definition,
- the return value of the function depends on the result of the recursive calls.

For pure computations, the above rules are sufficient. However, Erlang functions are often impure [18]. Therefore, the concept of component hygiene should be introduced [1]. Here the recursive calls must be possible to run in parallel, and hence their side effects, if there are any, must not be conflicting. For simplicity, we disregard this issue in this paper. Information on hygiene analysis can be found in [1].

To formally define the rules for *d&c* identification, we consider a function f, as well as the control-flow graph $G_{CF}(f)$ and dependence graph $G_D(f)$ built from f as a starting point. If the following conditions hold, f will be identified as a *d&c* candidate.

1. f must be recursive: it has an execution path which contains a call to itself;

$$\exists p \in EP(start_f),\ \exists c \text{ such that } call_f^c \in p$$

2. f must have a base case: it has an execution path which does not contain a call to itself;

$$\exists p \in EP(start_f) \text{ such that } (\nexists c : call_f^c \in p) \wedge (end_f \in p)$$

3. f must have multiple recursive calls in its body, as described by the following three possibilities.

- It may contain an execution path that contains at least two independent recursive calls[2]:

$$\exists c_1, c_2, \exists p \in EP(ret_f^{c_1}) \text{ such that } call_f^{c_2} \in p \wedge \forall a \in \mathrm{ARG}(c_2) : \neg(a \overset{\mathbf{dep}}{\rightsquigarrow} ret_f^{c_1})$$

  where ARG is the set of nodes representing the arguments of a function call.
- It may have an execution path containing a list comprehension with head expression h, which calls f directly or indirectly, that is:

$$\exists p \in EP(h),\ \exists c \text{ such that } call_f^c \in p.$$

- It may (directly or indirectly) call a *farm candidate* (see below) function g, which in turn calls f in its every recursive execution paths.

$$\exists p \in EP(start_f), \exists c_1, \exists g \text{ recursive function such that } call_g^{c_1} \in p \wedge$$

$$\forall q \in EP(start_g) : (\exists c_2 :\ call_g^{c_2} \in q) \rightarrow (\exists c_3 :\ call_f^{c_3} \in q)$$

Farm candidates. The above rules refer to another important class of parallelizable functions: those functions, which can be turned into a parallel task farm. Such farm candidates operate on a collection (set, list or stream) of data by applying a computation on each data item independently of the others. A formal characterization of *map-like functions* has been published in [1]. According to this characterization, a function f is map-like, if it satisfies the following conditions. It has a list parameter L, and returns a list. The head of the returned list may depend on the head of L, but may not depend on the tail of L. Similarly, the tail of the returned list may not depend on the head of L: it should simply be the result of a recursive call to f on the tail of f. Moreover, f may have further parameters, but all these parameters must be passed to the recursive call unchanged.

Note that the standard *map* function in Erlang, `lists:map/2`, satisfies these conditions; it is indeed a map-like function.

The characterization of map-like functions can be generalized to cover tail-recursive implementations of the same behaviour. Another generalization can cover the case when the input is a stream of data items (generated or `receive`d concurrently to the processing of those items). These generalizations are described in [22]. Further generalizations can also be made, and hence even more farm candidates can be identified – this way the scope of the *d&c* identification analysis also extends.

The rules given in this section are based on the studies presented in Section 4, and are therefore suitable to identify the *d&c* pattern candidates, and discard the non-patterns, as explained in that section.

[2] Obviously, there may be more than two recursive calls, but only the independent recursive calls result in exploitable parallelism.

4.3 Efficient approximation of *d&c* identification

The above rules are all based on the concept of EP, the execution paths in the control-flow graph. The computation of all execution paths is extremely time-consuming, which can turn *d&c* analysis impractical for larger code bodies. Therefore less precise, but more efficiently computable conditions have to be applied for such cases. Execution paths are used by our analysis for finding certain function calls. One can substitute this analysis with a similar one working on the functional call graph, G_{FC}. This is clearly an approximation, since G_{FC} only records which functions call which other functions, and does not allow us differentiate calls on different execution paths. Let the edges of G_{FC} be denoted with the *funcall* label, and a path in G_{FC} with *funcall+* (which is an edge in the transitive closure of G_{FC}).

Our *d&c* discovery analyses G_{FC}, and searches for patterns expressing iterative evaluation of a function. Figure 1 shows such a call graph pattern. If a function f calls (directly or indirectly) a recursive function g (i.e. g calls itself directly or indirectly), and g calls f, then our analysis suspects that f is called in every recursive step, so it is called multiple times by g.

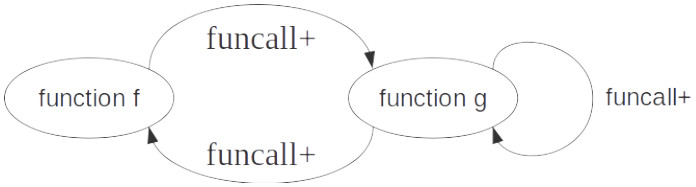


Fig. 1. A fragment in a function call graph typical for *d&c*

After collecting such suspicious f-g pairs, a more precise analysis can be applied to determine whether f is really a *d&c* candidate.

5 APPLICABILITY OF *D&C* DISCOVERY

This section reports on our findings on the applicability of *d&c* pattern candidate discovery. The analyses presented in this paper have been implemented within the ParaPhrase Refactoring Tool for Erlang. In order to investigate the effectiveness of our approach, we have analysed some small examples, some use cases developed within the ParaPhrase project, and some open source projects as well. Our findings were reported in ParaPhrase project deliverable D6.6 [21]. Here we demonstrate two applications in which several *d&c* candidates were found.

We verified manually (some of) the pattern candidates found by the tool, to see whether they really possess the properties that qualify them for the identified pattern. Most candidates (list comprehensions and applications of predefined higher-order functions) are trivially appropriate, hence we checked them only by random sampling. Map-like functions and divide-and-conquer algorithms are more sophisticated to characterize and harder to find, and so we paid more attention

to them. We checked the dozen map-like functions the tool found, but used only random sampling for the about one hundred divide-and-conquer candidates. Two interesting *d&c* candidates are presented below in Section 5.2.

5.1 Discovery statistics

First of all, we have analysed the source code of the distributed database management system Mnesia [25], which is part of the standard Erlang/OTP library. The analyzed code body contains 1,693 function definitions in 31 files, and consists of 22,653 effective lines of code. We could find 57 *d&c* candidates (Table 1).

Candidate	**Number of occurrences**	**Kind of pattern**
various library calls	72	farm
various library calls	36	reduce
list comprehension	58	farm
map-like function	5	farm
d&c-like function	57	divide and conquer

Table 1. Mnesia

Then we have analyzed some components of the RefactorErl tool [26] as well. The analyzed `referl_core` component contains 1,534 function definitions in 53 files, and consists of 19,694 effective lines of code. We could find 31 *d&c* candidates (Table 2).

Candidate	**Number of occurrences**	**Kind of pattern**
various library calls	139	farm
various library calls	55	reduce
list comprehension	347	farm
map-like function	3	farm
d&c-like function	31	divide and conquer

Table 2. RefactorErl: referl_core

5.2 Examples of interesting candidates

During the validation of pattern candidate discovery, we have encountered really nice instances of map-like and divide-and-conquer definitions. Here we point out two demonstrative cases.

The first example (Figure 2) shows a beautiful instance of the divide-and-conquer pattern: the `refcore_callanal:listcons_length/2` function almost completely follows the "canonical form" of *d&c*. It operates on a list; it splits the list in the divide-phase using `lists:partition/2`, it applies (through `listcons_length/1`)

itself iteratively with `lists:map/2` in the non-base case, and finally combines the results explicitly (with `lists:append/1`). The particularity of this example is that the *d&c* function is not directly recursive: it calls itself indirectly through another function.

```
1  listcons_length(ListExpr) ->
2    listcons_length(ListExpr, ?Graph:data(ListExpr)).
3  ...
4  listcons_length(N, #expr{}) ->
5    Ns = ?Dataflow:?reach([N], [back], true),
6    L1 = [N2 || N2 <- Ns, N2 /= N,
7                ?Graph:class(N2) == expr],
8    {L2, L3} = lists:partition(fun is_cons_expr/1, L1),
9    if L2 == [] orelse L3 /= [] ->
10         incalculable;
11      true  ->
12         lists:append(lists:map(fun listcons_length/1, L2))
13   end;
14 ...
```

Fig. 2. Nice *d&c* in RefactorErl

The second example (Figure 3) is again from the code of RefactorErl, namely the `refcore_pp:realtoken_neighbour/3` function. At a first sight, this function is a simple recursive function calling itself in line 16. If we further investigate this function, we find another execution path that calls `realtoken_neighbour_/4`. This function is a recursive function that calls `realtoken_neighbour/3` in its every recursive execution path. This is exactly the pattern in the function call graph which can be detected by our faster *d&c* candidate discovery algorithm.
Our discovery analysis identifies `realtoken_neighbour/3` as a divide-and-conquer definition, because it calls `realtoken_neighbour_/4`, which iterates through the `Parents` list, and calls `realtoken_neighbour/3` in each step. Note that the identification of the base condition (and the corresponding base function) of this *d&c* candidate is not straightforward, since the condition is scattered over several case expression headers and patterns. Hence the automatic transformation of this definition into a *d&c* pattern is indeed a challenge.

6 RELATED AND FUTURE WORK

Various approaches have been published related to parallel pattern identification. Some of these methods use purely static information, but others monitor the dynamic behaviour of the system as well.

In [1] a formal definition of "map-like functions" is given in order to automatically discover list-based elementwise computations. The paper also describes a variety of program shaping transformations to refactor the map-like pattern candidates in a syntactic form that can be then transformed into an application of the *farm* skeleton.

```
realtoken_neighbour(Node, DirFun, DownFun) ->
  case lists:member(?Graph:class(Node),
                    [clause,expr,form,typexp,lex]) of
    false -> no;
    _ ->
      case ?Syn:parent(Node) of
        [] -> no;
        [{_,Parent}] ->
          case lists:dropwhile(
                       fun({_T,N}) -> N/=Node end,
                       DirFun(?Syn:children(Parent))
                   ) of
            [{_,Node},{_,NextNode}|_] ->
              DownFun(NextNode);
            _ ->
              realtoken_neighbour(Parent, DirFun, DownFun)
          end;
        Parents ->
          realtoken_neighbour_(Parents, DownFun(Node),
                               DirFun, DownFun)
      end
  end.

% Implementation helper function for realtoken_neighbour/3
realtoken_neighbour_([], _FirstLeaf,_DirFun,_DownFun) ->
  no;
realtoken_neighbour_([{_,Parent}|Parents],
                     FirstLeaf, DirFun, DownFun) ->
  case realtoken_neighbour(Parent, DirFun, DownFun) of
    FirstLeaf ->
      realtoken_neighbour_(Parents, FirstLeaf,
                           DirFun, DownFun);
    NextLeaf  ->
      NextLeaf
  end.
```

Fig. 3. Really complex *d&c* in RefactorErl

In [2] the ParaPhrase Refactoring Tool for Erlang was introduced. This tool provides pattern discovery, candidate ranking based on performance measurements and estimates, as well as semi-automatic pattern introduction by refactorings. The cost models used by the tool were defined in [4]. Our presented *d&c* candidate discovery method extends this framework – although candidate ranking based on speedup estimates is not implemented yet for *d&c* (pattern discovery reports currently all *d&c* candidates without evaluating their parallel speedup potential).

The `skel` library [24] provides a set of reusable algorithmic skeleton implementations for Erlang. In [19] some high-level pattern implementations are presented as an extension to `skel`, including *d&c*. As future work, we plan to implement *d&c* transformations as well. To achieve our goal, we can rely on this high-level skeleton library. Two useful features of this library are that we can limit the number of started parallel processes, and force sequential evaluation where beneficial.

Although `skel` does not support distributed skeletons such as D-Clean [29], it is possible to extend it based on the Erlang concepts.

The Eden skeleton library [12] offers even more advanced *d&c* pattern implementations, for instance, the one based on the distributed expansion scheme. The three main properties of this implementation are: (1) a branching degree is given by a parameter representing the number of subproblems; (2) the process creation and allocation is controlled by a ticket list; and (3) every process keeps a subproblem for local evaluation. We are not aware of any pattern discovery tools for Eden, but it is definitely possible to implement one based on the approach of this paper. Moreover, it is also possible to re-implement Eden's sophisticated *d&c* patterns for Erlang.

Several other researches focus on efficient *d&c* implementations, including for example, Hermmann [9]. Note that the refactoring-centric approach of ParaPhrase (and hence PaRTE) fosters experimentation with the various implementations and parameter values, including thresholds controlling the parallel-sequential balance.

Although discovering *d&c* candidates is an interesting problem by itself, it becomes the most beneficial when a tool introduces parallelism (semi)automatically to the source code by replacing the sequential *d&c* with its parallel equivalent. In this case two problems have to be addressed: the efficiency of the parallel *d&c* (and hence the usefulness of parallelizetion) has to be investigated, and a transformation framework has to be capable of introducing the parallel *d&c* implementation.

In [6] an automated transformation framework was introduced to transform sequential *d&c* algorithms to parallel equivalents. The transformation uses a few annotations (which are required to be provided by the programmer) to identify the places where parallelism should be introduced.

The automatic transformation of *d&c* candidates was motivating our research, but is not covered in this paper. We set out this research topic for future work, based on the above mentioned papers.

Optimizing compilers are able to identify parallelizable code fragments, and also to parallelize them automatically. Of course, such transformations are, and should be, ultra-conservative, not allowing to change the semantics of the code. Our approach, in the contrary, can be more flexible and more effective, since we have a human in the loop: the final decision on parallelization is always made by our tool user. To mention but one parallelizing compiler, SkelML [13] is a parallel skeleton-based compiler for SML. It can automatically identify applications of certain higher-order functions as pattern candidates and transform them to applications of equivalent parallel skeletons. For instance, SkelML can transform *fold* function applications to application of *d&c* skeletons. However, the pattern discovery technique applied by SkelML is less generic than ours. Our tool is able identify various syntactic forms of pattern candidates which are not necessarily just applications of special higher order functions. Indeed, the strength of our tool is the analysis of recursive function calls.

Similar parallelization techniques have been developed for non-declarative languages as well. The tool AutoFutures [14], for example, uses static analysis of Java programs to discover source code fragments that have no data dependencies, and

can be candidates for parallelization.

The static analysis approach that we also follow can be replaced (or combined) with dynamic analyses as well. One can examine execution traces, like the tool [7] based on JavaSlicer, which uses dynamic dependence graphs to identify independent program paths in Java code, and recommends code fragments with a high potential for parallelization. One can also integrate a static analysis based approach into one using run-time monitoring of executions, which we plan for future work.

7 CONCLUSION

This paper investigated the concept of divide-and-conquer pattern discovery, a static program analysis technique to automatically identify parallelizable code fragments. This analysis can cope with the many syntactically different occurrencies of *d&c* behaviour. Depending on how conservative the analysis is, the technique can be exploited in an optimizing compiler, or in an integrated software development environment, which supports refactoring. We focussed on the latter use case. This allows our analysis to find *d&c* candidates which are too hard to transform automatically into a skeleton-based parallel pattern implementation – in an IDE the user may indeed be able to carry out the transformation manually, or semi-automatically. Since the final decisions are made by the user, this approach tolerates false positives, and hence the analysis can be parametrized (and the discovery tool be dynamically configured) by the level of conservativeness and that of aggressivity. This allows the user to make the necessary practical trade-offs between effectiveness and safety, and between effectiveness and latency.

Our analysis is built on top of control-flow, data-flow, reaching, function-call, and control-dependence analyses. Therefore, the presented *d&c* candidate discovery is applicable for a wide range of programming languages. However, we have worked out the details of the analysis for the Erlang language, and for this the paper summarized the Erlang-specific details of the CFG and DFG construction. We have implemented the analysis in a software development tool, the ParaPhrase Refactoring Tool for Erlang.

The numerous examples shown in the paper are also written in Erlang. As a final evaluation of the approach, we applied the analysis on real-world open-source code bases.

The lessons learned from the presented research are the following. Divide-and-conquer algorithms are often implemented using recursion: indeed, we have defined *d&c* as a function which calls itself recursively multiple times. Therefore, a static analysis of (direct and indirect) recursive calls is able to detect (many) occurrences of *d&c*. Such an analysis can be defined in the presence of an interprocedural control-flow and data-flow analysis, and hence it is language and paradigm independent. In an imperative language, for instance, a *d&c* function may include a loop construct with a body containing a recursive call. Of course, it is possible to implement a *d&c* algorithm without recursion, for instance using only iteration (loops in an imperative

language) and a stack data structure – such an implementation will not be found by the proposed analysis.

The analysis of recursive calls can be defined using the concept of execution paths. It turned out that this analysis is rather time-consuming. However, a faster, though less precise, analysis based on the function call graph also performs very well in practice.

Acknowledgement The research presented in this paper was supported by the Seventh Framework Programme (FP7) under the contract number: 288570.

REFERENCES

[1] Bozó, I., Fördős, V., Horpácsi, D., Horváth, Z., Kozsik, T., Kőszegi, J., and Tóth, M.: Refactorings to enable parallelization. In *Trends in Functional Programming, 15th International Symposium*, Lecture Notes in Computer Science, Vol. 8843, pages 104–121. Springer International Publishing, 2015.

[2] Bozó, I., Fördős, V., Horváth, Z., Tóth, M., Horpácsi, D., Kozsik, T., Kőszegi, J., Barwell, A., Brown, C., and Hammond, K.: Discovering parallel pattern candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, pages 13–23. ACM, New York, NY, USA, 2014.

[3] Bozó, I., Horpácsi, D., Horváth, Z., Kitlei, R., Kőszegi, J., Tejfel, M., and Tóth, M.: RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proc. 12th Symposium on Programming Languages and Software Tools*, pages 138–148, 2011.

[4] Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., and Elliott, A.: Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming* 42(4):564-582, 2014.

[5] Cole, M.: *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA, 1991.

[6] Freisleben, B. and Kielmann, T.: Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *COMPUTING AND INFORMATICS* 14(6):579–596, 1995

[7] Hammacher, M., Streit, K., Hack, S., and Zeller, A.: Profiling Java Programs for Parallelism. In *Proc. ICSE Workshop on Multicore Software Engineering*, pages 49–55, 2009.

[8] Hammond, K., Aldinucci, M., Brown, C., Cesarini, F., Danelutto, M., González-Vélez, H., Kilpatrick, P., Keller, R., Rossbory, M., and Shainer, G.: The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, Vol. 7542, pages 218–236. Springer Berlin Heidelberg, 2013.

[9] Herrmann, C.A.: *The Skeleton Based Parallelization of Divide and Conquer Recursions.* Logos-Verlag, 2001. ISBN 9783897225565.

[10] Horpácsi, D., and Kőszegi, J.: Static analysis of function calls in Erlang. *e-Informatica Software Engineering Journal* 7:65–76, 2013.

[11] Logan, M., Merritt, E., and Carlsson, R.: *Erlang and OTP in Action*, Manning Publications Co., 2010. ISBN 9781933988788.

[12] Loogen, R.: Eden – parallel functional programming with Haskell. In *Central European Functional Programming School*, Lecture Notes in Computer Science, Vol. 7241, pages 142–206. Springer Berlin Heidelberg, 2012.

[13] Michaelson, G., Ireland, A., and King, P.: Towards a Skeleton Based Parallelising Compiler for SML. In *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, 1997.

[14] Molitorisz, K., Schimmel, J., and Otto, F.: Automatic Parallelization Using Autofutures. In *Proc. 2012 Int'l Conf. on Multicore Software Engineering, Performance, and Tools*, Lecture Notes in Computer Science, Vol. 7303, pages 78–81. Springer Berlin Heidelberg, 2012.

[15] Mou, Z. G., and Hudak, P.: An algebraic model for divide-and-conquer and its parallelism. *Journal of Supercomputing* 2(3), 1988.

[16] Muchnick, S. S.: *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

[17] Nielson, F., Nielson, H. R., and Hankin, C.: *Principles of Program Analysis.* Springer, 1999, corrected 2005.

[18] Pitidis, M., and Sagonas, K.: Purity in Erlang. In *Proc. 22nd Int'l Conf. on Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, Vol. 6647, pages 137–152. Springer Berlin, Heidelberg, 2011.

[19] ParaPhrase project WP2.: Initial Implementation of Application-Specific Patterns. Technical report, University of Pisa, September 2013.

[20] ParaPhrase project WP4.: Specification of Pattern Candidate Refactoring Rules. Technical report, ELTE-Soft Nonprofit Ltd., July 2014.

[21] ParaPhrase project WP6.: Final Report on Experimental Evaluation. Technical report, University of Stuttgart, March 2015.

[22] ParaPhrase project WP2.: Final Pattern Discovery. Technical report, ELTE-Soft Nonprofit Ltd, March 2015.

[23] Shivers, O.: Control flow analysis in Scheme. In *Proc. SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pages 164–174, 1988.

[24] Skel Tutorial. Available at `http://chrisb.host.cs.st-andrews.ac.uk/skel-test-master/tutorial/bin/tutorial.html`, 2014.

[25] Source code of Mnesia. `https://github.com/erlang/otp/tree/maint/lib/mnesia/src`

[26] Source code of RefactorErl. `http://plc.inf.elte.hu/erlang/dl/refactorerl-0.9.14.09.zip`

[27] The ParaPhrase project. `http://www.paraphrase-ict.eu`, 2014.

[28] Tóth, M., and Bozó, I.: Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School*, Lecture Notes in Computer Science, Vol. 7241, pages 440–498. Springer, 2012.

[29] Zsók, V., Hernyák, Z. and Horváth, Z.: Designing Distributed Computational Skeletons in D-Clean and D-Box. In *Central European Functional Programming School*, Lecture Notes in Computer Science, Vol. 4164, pages 223–256 Springer, 2006.

Tamás Kozsik is associate professor at Eötvös Loránd University (Budapest, Hungary), where he is vice dean for projects and innovation at the Faculty of Informatics. He teaches programming paradigms and languages. His research is focusing on language technologies (including type systems, domain specific languages, source code analysis and transformations, parallel programming, and formal methods). Recently he was Principal Investigator in the Parallel Patterns for Adaptive Heterogeneous Multicore Systems (ParaPhrase) EU FP7 project.

Melinda Tóth works as a researcher at ELTE-Soft Nonprofit Ltd. (Budapest, Hungary), leading the ELTE-Ericsson Software Technology Lab. She is also assistant lecturer at Eötvös Loránd University, teaching distributed systems and Erlang/OTP technology. Melinda Tóth is chief architect of RefactorErl, a static source code analysis and transformation system for Erlang. She co-chairs ACM SIGPLAN Erlang Workshop in 2015 and 2016.

István Bozó is researcher at ELTE-Soft Nonprofit Ltd. (Budapest, Hungary), and assistant lecturer at Eötvös Loránd University. His main research topic is impact analysis of functional programs based on control-dependence graphs. He is working on static program analysis and refactoring in the RefactorErl and the ParaPhrase projects. István Bozó teaches functional programming as well as formal methods for distributed systems.

Zoltán Horváth is project manager at ELTE-Soft Nonprofit Ltd. (Budapest, Hungary), and full professor at Eötvös Loránd University, heading the Programming Languages and Compilers Department. He is also dean of Faculty of Informatics, and Director of the EIT Digital Budapest Associate Partner Group. He is teaching and researching functional programming and formal methods for distributed systems. Zoltán Horváth supervised numerous national and international projects, among others he was Principal Investigator in the Parallel Patterns for Adaptive Heterogeneous Multicore Systems (ParaPhrase) EU FP7 project.

Adat és kiértékelési függőségi elemzés funkcionális nyelvekre – Erlang programok statikus elemzése

A szoftverfejlesztést támogató eszközök jelentősége rohamosan nő az utóbbi évtizedekben. A forráskódok mérete akkorára nő, hogy humán erővel átlátni szinte lehetetlen, de legalábbis nehézkes és időigényes folyamat. Így egyre inkább elterjednek azok az eszközök, melyek a kód megértést, karbantartást, hibakeresést támogatnak vagy éppen lehetőséget nyújtanak a forráskód különböző szempontok szerinti refaktorálására. Ez a támogatás történhet dinamikusan, azaz futási időben, illetve statikusan, azaz fordítási időben. Előbbi esetben a már futó szoftver monitorozásával, esetlegesen a kód instrumentálásával nyerhetünk ki információt és segíthetjük ezzel a fejlesztőket. Utóbbi esetben viszont nincs szükség a szoftver futtatására, csupán a forráskód alapján gyűjtünk információt és használjuk fel különböző célokra. Dolgozatomban ez utóbbi módszerrel, a forráskódok statikus elemzésével foglalkoztam, Erlang nyelvhez.

Definiáltam Erlang programok elsőrendű adatfolyam-gráfját, és az ezen a gráfon értelmezett elsőrendű adatfolyam relációt. A RefactorErl keretrendszer alap eszközkészletéhez igazítva megadtam az ehhez tartozó algoritmusokat. Erlang programok adatfolyam-gráfját így inkrementálisan fel tudjuk építeni a forráskódok változásának figyelembe vételével. A adatfolyam reláció segítségével pedig választ kaphatunk olyan kérdésekre, hogy mi lehet a program egy adott pontján lévő kifejezésének az értéke, illetve hogy egy adott érték milyen programpontokra juthat el. A reláció interprocedurálisan számítható és figyelembe veszi a hívási kontextust. Az adatfolyam relációt felhasználásával megadtam, hogyan számítható ki az aszinkron üzenetküldések és -fogadások közötti közvetlen adatfolyam.

Definiáltam Erlang programokon értelmezett adatfüggőség relációt, mely az adatfolyam-gráf kiterjesztésén a viselkedésfüggőség gráfon számítható ki. A függőségi reláció megadja, hogy két Erlang programbeli kifejezés között van-e függés, azaz kiértékelésük függ-e egymástól. A RefactorErl keretrendszerhez kidolgozott algoritmusok felhasználásra kerültek olyan problémák megoldásában mint a releváns tesztesek kiválasztása, vagy párhuzamosítható komponensek függőségi kapcsolatainak az ellenőrzése.

Az adatfolyam, adatfüggőség és egyéb statikus elemzések felhasználásával kidolgoztam különböző jól párhuzamosítható számítási modellek viselkedésének leírását, mint például az elemenkénti feldolgozás. A megadott szabályok alapján a RefactorErl keretrendszerben megadhatók azok a mintafelismerési algoritmusok, melyek segítségével azonosíthatóak azok a szekvenciális kódrészletek, melyek lecserélhetőek egy ekvivalens párhuzamos végrehajtásra.

Data flow and dependence analyses for functional languages – Static analysis of Erlang programs

The importance of the tools to support software development is increasing. The size of the software products makes manual scanning for certain information almost impossible, or time consuming. Therefore tools to support code comprehension, development, maintenance, debugging, or even automatic source code transformation are really desired. We can distinguish dynamic and static tools. The former analyses the software at runtime by monitoring, instrumenting the code. The latter analyses the source code itself without executing the program. In my thesis I have developed new static analyses methods for the Erlang programming language to support code comprehension and further static analyses.

I have defined the first order data flow graph for Erlang programs, and the data flow relation among the nodes of the graph, the first order data flow reaching. The reaching relation itself is able to identify the possible values of an expression at some point in the program. It also identifies the expressions where a certain value may flow. Based on the definitions I have introduced the data flow graph building and reaching calculation algorithms using the RefactorErl framework. The data flow graph building algorithm is incremental, therefore the changes of the source code can be handled without reanalysing the whole software. The data flow reaching algorithm is interprocedural and aware of function call context. Using data flow reaching I have defined the direct data flow among asynchronous message sending and receiving expressions.

I have defined a data dependence relation among Erlang expressions based on the behaviour dependence graph that is an extension of the data flow graph. The dependence relation defines whether two expressions from the Erlang source code depends on each other. The corresponding algorithms have been defined in the framework of RefactorErl, and have been used in further static analyses, namely in change impact analysis and pattern discovery.

I have defined the behaviour of parallelisable computations, such as the elementwise processing. The definitions use the studied data flow, data dependence relations and other static analysis methods, such as control flow graphs and execution paths. Using the definitions we can identify sequential code fragments that can be replaced with parallel equivalents. The pattern discovery algorithms have been defined in the RefactorErl framework.

[1]ADATLAP
a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai

A szerző neve: Tóth Melinda

MTMT-azonosító: 10029387

A doktori értekezés címe és alcíme: Adat és kiértékelési függőségi elemzés funkcionális nyelvekhez, Erlang programok statikus elemzése

DOI-azonosító[2]: 10.15476/ELTE.2018.171

A doktori iskola neve: Informatika Doktori Iskola

A doktori iskolán belüli doktori program neve: Az informatika alapjai és módszertana

A témavezető neve és tudományos fokozata: Dr. Horváth Zoltán, PhD, habil

A témavezető munkahelye: Eötvös Loránd Tudományegyetem, Informatikai Kar

II. Nyilatkozatok

1. A doktori értekezés szerzőjeként[3]

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét, Kulcsár Adinát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[4]

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (*dátum*)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[5]

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.[6]

2. A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: Budapest, 2018.08.30.

a doktori értekezés szerzőjének aláírása

---

[1] Beiktatta az Egyetemi Doktori Szabályzat módosításáról szóló CXXXIX/2014. (VI. 30.) Szen. sz. határozat. Hatályos: 2014. VII.1. napjától.

[2] A kari hivatal ügyintézője tölti ki.

[3] A megfelelő szöveg aláhúzandó.

[4] A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.

[5] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.

[6] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.